Partial Satisfaction Planning: Representation and Solving Methods

by

J. Benton

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved July 2012 by the
Graduate Supervisory Committee:

Subbarao Kambhampati, Chair
Chitta Baral
Minh B. Do
David E. Smith
Pat Langley

ARIZONA STATE UNIVERSITY

August 2012

ABSTRACT

Automated planning problems classically involve finding a sequence of actions that transform an initial state to some state satisfying a conjunctive set of goals with no temporal constraints. But in many real-world problems, the best plan may involve satisfying only a subset of goals or missing defined goal deadlines. For example, this may be required when goals are logically conflicting, or when there are time or cost constraints such that achieving all goals on time may be too expensive. In this case, goals and deadlines must be declared as soft. I call these partial satisfaction planning (PSP) problems. In this work, I focus on particular types of PSP problems, where goals are given a quantitative value based on whether (or when) they are achieved. The objective is to find a plan with the best quality.

A first challenge is in finding adequate goal representations that capture common types of goal achievement rewards and costs. One popular representation is to give a single reward on each goal of a planning problem. I further expand on this approach by allowing users to directly introduce utility dependencies, providing for changes of goal achievement reward directly based on the goals a plan achieves. After, I introduce time-dependent goal costs, where a plan incurs penalty if it will achieve a goal past a specified deadline.

To solve PSP problems with goal utility dependencies, I look at using state-of-the-art methodologies currently employed for classical planning problems involving heuristic search. In doing so, one faces the challenge of simultaneously determining the best set of goals and plan to achieve them. This is complicated by utility dependencies defined by a user and cost dependencies within the plan. To address this, I introduce a set of heuristics based on combinations using relaxed plans and integer programming formulations. Further, I explore an approach to improve search through learning techniques by using automatically generated state features

to find new states from which to search. Finally, the investigation into handling time-dependent goal costs leads us to an improved search technique derived from observations based on solving discretized approximations of cost functions.

For my mother Linda Susan Kamins and

late father John “White Cloud” Eldon Benton, Ed.D.

ACKNOWLEDGEMENTS

This dissertation exists because of the support I received from the people in my life. Academically, the work I have done rests firmly on the shoulders of giants. It builds on past work, and required mountains of discussion, advice and collaboration with numerous individuals both within Arizona State University and outside of it. On a more personal level, my family, friends, dissertation committee, colleagues and labmates have all graciously helped me when they have seen I needed it.

Foremost, I must thank my advisor, Subbarao Kambhampati, who helped me to keep things in perspective with his amazing ability to simultaneously see the details and the big picture in everything. Rao's vast knowledge and thorough understanding of automated planning has helped me in immeasurable ways and his good humor, strong willed advice, technical know-how, long-term patience and outspoken nature have guided me through the process of research. The dedication he gives to his students is unwavering and kept me on track in pursuit of my research goals throughout various bumps in the road.

All of my other committee members were also invaluable. Chitta Baral gave me great technical insights into KR during his amazing lectures. Minh Do provided priceless advice and technical insights and acted as both a friend and a mentor. David Smith asked great, tough questions and passed along important wisdom on using planning technologies in the real world. Pat Langley imparted his broad perspective on artificial intelligence and cognative science and has always given excellent advice.

I have had many collaborators in my research that have contributed in unique and important ways. They all have been vital to my research and each deserves special mention and high praise for their advice, mentorship and contributions. First, I must thank those who sit in my lab and have worked with me directly in person:

Subbarao Kambhampati (my advisor), Menkes van den Briel, William Cushing, Tuan Nguyen, Sungwook Yoon, and Kartik Talamadupula. I have also worked with many individuals in other locations, both near and far: Patrick Eyerich and Robert Mattmüller (University of Freiburg); Andrew Coles and Amanda Coles (King's College London); Malte Helmert (Basal University); Sophia Kelley and M. Scott Thompson (Arizona State University in Anthropology); Matthias Scheutz (Tuffs University); Rehj Cantrell and Paul Schermerhorn (Indiana University); Wheeler Ruml, Ethan Burns, Sofia Lemons, Allen Hubbe, and Jordan Thayer (University of New Hampshire); and Minh Do (NASA Ames Research Center), who is both a committee member and a collaborator. To all these people—you all know how important you were and many thanks!

Others in the automated planning community that have given me great insights and perspectives include Patrik Haslum, Maria Fox, Derek Long, Alan Fern, Ronen Brafman, Carmel Domshlak, Hécter Geffner, Mausam, Blai Bonet, Daniel Weld, Daniel Bryce, Jörg Hoffmann, Jorge Baier, Emil Keyder, Héctor Palacios, Christian Fritz, Sheila McIlraith, Sylvie Thiébaux, Dana Nau, Ugur Kuter, Robert Goldman, Alfonso Gerevini, Jeremy Frank, Adi Botea, Erez Karpas, Rong Zhou, Michael Katz, Gabriele Röger, Peter Gregory, Silvia Richter, Manuela Veloso, Kanna Rajan, David Musliner, Terry Zimmerman, Stephen Smith, Adele Howe, Saket Joshi, Tran Cao Son, Angel Garcia-Olaya, Stefan Edelkamp, Sven Koenig, Richard Russell, Romeo Sanchez, Martin Müller, Hootan Nakhost, Richard Dearden, Marie desJardin, Michael Moffett, Alastair Andrew, Bram Ridder, Neil York-Smith, Ian Little, and Håkan Younes.

I must also thank my closest friend, Gretchen Corey, who endured my whining and gave me unique perspectives as I pushed through my research; my mom, Linda

Kamins, and late father, John Benton, who always provided an open ear; and finally, my dog, Molly, who was always happy to see me and forgiving of her long nights home alone.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

Chapter 1
Introduction

Research into automated planning has usually focused on the full achievement of all goals. But this approach neglects many fundamental real-world scenarios where goals and their achievement deadlines can be only partially satisfied. For example, goals might be logically conflicting, and resource constraints may prevent their timely achievement. Consider Mars rover mission planning. In this situation, goals involve performing experiments at a variety of locations with cost constraints (e.g., battery power), making it so deadlines might be missed or only a subset of the goals can be satisfied [88]. We call these problems *partial satisfaction planning* (PSP) problems. In this dissertation, we will focus on particular types of PSP problems, such that goal achievement can be given some value (e.g., reward) and actions are given costs. The objective is to balance a goal's achievement value with action costs to achieve the best plan. In the case where we assign rewards to goals, we want to maximize the overall difference between the reward gained for achieving goals and the cost of the actions to achieve them, or *net benefit* [95, 88].

In tackling partial satisfaction planning, we face dual challenges:

- Finding adequate goal reward representations that capture common types of goal achievement reward and goal deadline cost; and
- Finding effective methods to solve planning problems that have goals with these representations.

Representations and solving methods have a strong interaction with one another and decisions made for approaching one challenge have a direct effect on the other.

For instance, focusing on a general representation of goal achievement reward directly effects (and often increases) the difficulty of solving planning problems that involve those representations. Hence, the two topics fold into one another and separating representations and solving methods becomes convoluted. In the end, we chose to look at our representations in terms of their generality; we reasoned that more general representations would yield solving methods capable of handling less expressive representations that others may find more appealing from a user standpoint. Further, our solving methods may be applicable to other, related problems and be less specialized in nature.

Given our representations, we solve resulting partial satisfaction planning problems using state-of-the-art methods in automated planning. The decision to use these methods were based both on our own experiments and comparisons (which we discuss) and their applicability to the problems at hand. In the rest of this introduction, we summarize the representations and solving methods that we applied.

1.1 Representing Goal Achievement Rewards and Costs

As a baseline representation for goal reward, one can associate a single reward value with each goal fact. But even with this relatively simple representation, the process of finding goals on which to focus is complicated by the fact that they interact with one another. Actions may share in their achievement of goals (positive interaction) or conflict (negative interaction). These types of interactions introduce *cost dependencies* between goals because the cost of achieving them separately may differ from the cost of achieving them together.

This dissertation work further extends on this representation to directly address *utility dependencies* which allow users to specify changes in utility on sets of goals [29]. Two concrete examples of utility dependency are mutual dependency

and conditional dependency. For mutual dependency, the utility of a set of goals is different from the sum of the utility of each individual goal. For example, (1) while the utility of having either a left or right shoe alone is zero, the utility of having both of them is much higher (i.e., the goals complement each other); (2) the utility of having two cars is smaller than the sum of the individual utilities of having each one of them (i.e., the goals substitute each other). Conditional dependency is where the utility of a goal or set of goals depends on whether or not another goal or set of goals is already achieved. For example, the utility of having a hotel reservation in Hawaii depends on whether or not we have already purchased a ticket to Hawaii. A main representational challenge is in finding a model where the different types of goal utility dependencies can be naturally expressed. For this, we use the *Generalized Additive Independence (GAI)* model [2], combining utility theory and deterministic planning. This model has the advantages that it is expressive, general, and can be compiled from other models such as UCP-Networks [14].

We also define *time-dependent goal costs*, where no cost is given for achieving a goal by a deadline time, but after that time point cost increases until it reaches a maximum cost value. For example, consider a satellite where goals should be achieved on time to avoid negative impact to an organization's budget (due to employee and equipment usage). There exists a definable function on the cost for missing the satellite's goals. The main challenge in defining these types of goals is how to best represent them such that they can be easily solved. Using a cost function on goal achievement time, even if the function is linear, poses some particular challenges on how to limit the search space to enable solutions to be found efficiently. To these ends, we look at representing linear cost functions directly, as continuous functions over time, and using discrete approximations.

To model linear cost functions directly, we use a small subset of the planning domain description language PDDL+ [43], an extension of PDDL that allows the modeling of continuous processes over time. This provides for the ability to capture a numeric representation of the *current time* within a plan, a capability that is oddly lacking from other versions of PDDL. After this, we then define an action that "collects" the penalty cost based on when the goal is achieved, making the assumption that the goal can be achieved only once (though relatively simple extensions can remove this assumption).

For handling the discretized model, we turn to planning domain description language PDDL3 [48], which allows us to model soft deadlines with discrete penalities where if the deadline is missed, then a penalty is paid. Using this language, we define several deadlines for each original continuous deadline goal, generating a step function and allowing the approximation of the continuous cost function.

1.2 Solution Methods for Partial Satisfaction Planning

The main contribution of this dissertaiton is in solving these problems with goal utility dependencies, where users can define reward dependencies between goals; and time-dependent goal achievement costs, such that missing a deadline incurs some penalty cost. We also explore methods for compiling other partial satisfaction planning problem definitions into the *net benefit* model and look toward how to solve them.

Solving for Goal Utility Dependencies

To solve PSP problems with goal utility dependencies we introduce heuristics for an anytime, best-first branch and bound search (originally defined in the planner *Sapa*$^{\mathcal{PS}}$ [7]) and a learning approach that can be used to improve upon solutions by restarting the search. The heuristic methods use integer programming (IP) formula-

tions to solve the combinatorial problems associated with goal and action selection. The approach for improving search through learning techniques uses search state features to find new states from which to search.

In developing heuristics for partial satisfaction planning, the challenge faced is in simultaneously determining the best set of goals to achieve and finding the best plan for them. Both are complicated by utility and cost dependencies within a plan. We first introduce a set of heuristics that use a combination of cost propagation over a relaxed planning graph (similar to the one used in the planner FF [63]) and an IP encoding to capture goal achievement cost and goal utility [29]. Specifically, the approach solves a relaxed version of the planning problem that ignores negative interactions between actions and utility dependencies between goals. It then encodes the solution to the relaxed problem in an IP format, capturing the positive cost dependencies between actions and all goal utility dependencies. The solution to this IP encoding gives an inadmissible heuristic measure for states during search, which effectively removes goals from consideration that appear unreasonable to achieve. We call this heuristic h^{GAI}_{relax}. We also implemented an admissible version of this heuristic, which does not find a solution to the relaxed problem but instead uses a *max* propagation over the planning graph structure, capturing a lower bound on the cost to reach the goals. Then, having found that cost, it encodes the values along with the utility dependencies of the goals in an IP formulation whose solution provides an admissible heuristic we call h^{GAI}_{max}.[1]

As one would expect, these two heuristics perform much better than a heuristic that completely ignores goal utility dependencies and uses a solely procedural approach to removing goals (as done in the planner *Sapa*$^{\mathcal{PS}}$). Its performance also

[1] In the case of maximizing net benefit, an admissible heuristic will always over-estimate the net benefit of goal achievement.

scales much better than encoding the entire problem as a bounded-length integer program [29].

While the relaxed plan-based heuristics do a fair job of estimating the cost of goal achievement, ultimately one would like to select actions and goals together to optimize *net benefit*. This requires a heuristic estimate with more of an "optimization" perspective. A standard way of setting up a relaxation sensitive to this is to formulate an IP encoding for a problem, and then compute the linear programming (LP) relaxation of this encoding. In addition to being sensitive to the objectives of the optimization, such encodings are also sensitive to the negative cost interactions between actions—something that is notoriously missing in standard relaxed plan heuristics. A challenge in adopting such an approach involves deciding exactly what type of IP encoding to use. While bounded horizon encodings have been explored in past work [29], this can only guarantee feasible plans, and offers no guarantees of optimality.

Hence, we use a heuristic adopting a compact IP encoding that is not dependent on a horizon bound. It represents the causal interactions between actions, thereby taking negative interactions between actions into account. It is a relaxation of the original problem in that it ignores action ordering, allowing for fewer constraints and variables than typical encodings. By itself, this IP encoding gives an admissible heuristic. But to increase scalability, an LP relaxation of the encoding is used, keeping the heuristic admissible. We call this heuristic h^{GAI}_{LP}. On domains we tested, with the use of lookahead techniques, this heuristic performs quite a bit better than the h^{GAI}_{relax} heuristic (also applying similar lookahead techniques) in terms of plan quality given a bounded solving time [9].

Improving Plan Net Benefit Through Learning

Along with heuristics, this dissertation also investigates a method of improving heuristic values through learning techniques. With the optimization nature of PSP *net benefit* problems, the STAGE algorithm [15] looked to be an attractive methodology, as it had shown promise for improving search in the context of optimization. STAGE is an online learning approach that was originally invented to improve the performance of random-restart, hill-climbing techniques on optimization problems. Rather than resort to random restarts, which may or may not help the base search to escape a local minimum, STAGE aims to learn a policy that can intelligently generate restarts that are likely to lead the hill-climbing search towards significantly better local optima. The algorithm works in two stages: Stage 1, where a base-level hill-climbing search is run until reaching a local minimum and Stage 2, where the algorithm trains on a sequence of states that the hill-climbing search passed through. The second stage learns a function that predicts, for a given state s, the value v of the optima that will be reached from s by hill-climbing. This learned function is then used in a new local search to scout for a state s' that has more promise to reach an even better state. If the learner is effective, s' is expected to be a good restart point. This work adapts this approach to operate within our systematic (best-first branch and bound) search. We call our modified approach Stage-PSP.

The main challenge in adopting STAGE to PSP *net benefit* is in finding appropriate state features for the learner. Boyan and Moore [15] used *handcrafted* state features. Unfortunately, it is infeasible to hand-generate features for every planning domain and problem. Moreover, such interventions run counter to the tenets of domain-independent planning. Instead, the features should be generated automatically from the planning problems. This work uses two techniques for generating

features. The first uses the "facts" of the states and the actions leading to those states as features. The second uses a more sophisticated taxonomic syntax to generate higher level features. Both were implemented and tested them using our h^{GAI}_{relax} heuristic. The results show the promise of this type of learning approach, in one domain showing significant improvements over using the h^{GAI}_{relax} heuristic alone.

Solving for Related Partial Satisfaction Planning Models

Though PSP *net benefit* is one model of representing PSP problems, another broadly used model for PSP was introduced in the 2006 International Planning Competition. The competition organizers defined a language called PDDL3 (version 3 of the Planning Domain Description Language). In it, they introduced a myriad of features, including soft top-level goals that induced a cost if they were not satisfied. They generated subsets of PDDL3 for the competition, one of which was *simple preferences* (PDDL3-SP), and generated a competition track for this subset. We found that these problems can be compiled into PSP *net benefit* such that they can be solved by a PSP *net benefit* planner. Indeed, we implemented this compilation and entered a planner called *Yochan*$^{\mathcal{PS}}$ into the planning competition [7]. This planner compiles PDDL3-SP problems into PSP *net benefit* problems and solves them using the planner *Sapa*$^{\mathcal{PS}}$. The entry received a *distinguished performance* award. Later, we also tried a compilation of PDDL3-SP into cost-based planning in a planner called *Yochan*$^{\mathcal{COST}}$ which experiments performed worse than the compilation to PSP *net benefit*.

Solving for Time-Dependent Goal Cost

All of the solving methods discussed so far relate to handling atemporal goals. However, there also exists an important class of PSP problems that involve the partial satisfaction of deadlines. In these problems, a planner may find a plan that

achieves a goal past its stated temporal deadline, and given this the plan will incur a penalty cost dependent on when in the plan the goal is finally achieved. To solve for these types of problems, we make the assumption that cost is monotonically increasing and that all cost increases occur linearly to some maximum cost value. As mentioned earlier, we look at solving directly for the continuous representation of the cost function and a discretized version of the cost function. Solving for the discretized model yielded key insights and gave way to a tiered search approach, combining the efficiency benefits that the discretized model provides with the accuracy that the continuous model provides. All of the solving methods were implemented in a modified version of the state-of-the-art temporal planner POPF to create a planner called OPTIC (Optimizing Preferences and TIme-Dependent Costs).

In the continuous models we described, the planner was modified to parse and handle the extension allowing it to capture the *current time* within a plan. With the best-first branch-and-bound search process used in the POPF planner, the algorithm uses initial, candidate solutions to prune away the search space by using an admissible estimate on the plan cost to prune parts of the search space that we can guarantee will lead to worse solutions. For the discretized model, we use the built-in solving methods within OPTIC made for handling PDDL3 soft deadlines. The results show that various decretizations can do better than a continuous model, dependent on the domain. However, upon investigating the reason for this, it turns out that the reason the discretized models perform better is because the continuous models' direct representation of the cost functions provide less pruning ability than the discretized model. From these insights, we introduce a tiered search approach that searches for initial candidate solutions using pruning similar to that seen in the discretized models. With an initial candidate solution, the technique performs repeated searches

mimicking finer and finer grained discretized pruning, gradually approaching the search pruning found in the continuous model. This approach turns out to be overall superior than either directly handling discretized models or continuous models in the domains tested.

The rest of this dissertation proceeds as follows. We discuss how we formally represent goal rewards, and the extension into goal utility dependencies, plus our extension for time dependent goal rewards (Chapter 2). In Chapter 3, we discuss the anytime search used in our pursuit of solving PSP net benefit problems. We then discuss the technical details of heuristics and the learning approach for solving PSP *net benefit* problems with goal utility dependencies along with empirical results in Chapter 4. In Chapter 5 we discuss the compilation from PDDL3-SP to PSP *net benefit* and the results from an entry into the 5^{th} International Planning Competition in 2006, *Yochan*$^{\mathcal{PS}}$, that used this compilation. We also show a comparison against using a compilation to cost-based planning in the same planning system. Also in that section, we discuss difficulties faced when attempting to select goals up-front on problems from that planning competition. Chapter 6 discusses the investigation into solving planning problems with *time dependent goal costs*. Finally, Chapter 7 goes over related work and we conclude in Chapter 8.

Chapter 2

Representations for Partial Satisfaction Planning

Classic automated planning problems define an initial state, a conjunctive set of goals. The objective is to find a sequence of actions, also defined in the problem, that leads from the initial state to a state containing all of the goals. Partial satisfaction planning is planning where only some goals or constraints can be satisfied in full. It can be seen as a generalization of classical planning and provides a natural extension to capture a range of scenarios that involve limited resources. Those limits can force a choice to ignore goals or constraints that are desired but unnecessary. This means that a user must mark goals and other constraints as soft, or optionally achievable. Further, plans must have a ranking between them, because otherwise the natural procedure would be to simply ignore everything that is marked as soft. To enable this ranking, my work assigns quantitative functions over meeting soft goals and deadlines.

To start, classical planning is the problem of transforming an initial state I into a goal state $G \subseteq \mathcal{G}$, given a finite set of fluents F where $I \subseteq F$ and $\mathcal{G} \subseteq F$. To transform I into a state G containing all fluents of $\mathcal{G}$, we define a set of actions A, such that each action $a \in A$ has a set of preconditions, $pre(a) \subseteq F$, a set of delete effects, $del(a) \subseteq F$ and a set of add effects, $add(a) \subseteq F$. Applying an action a to a state s requires that $s \subseteq pre(a)$. When applied to s, an action a generates a new state s' such that $s' = (s \setminus del(a)) \cup add(a)$. The objective is to generate a plan, or a sequence of actions $P = (a_0, a_1, \ldots, a_n)$ such that applying each action in sequence leads to a state G where $\forall g \in \mathcal{G}, g \in G$.

We first look at partial satisfaction planning with net benefit, which extends on this. It is the problem of finding a plan with the maximum *net benefit* or difference between achieved goal reward and action costs [88, 95]. Each goal $g \in \mathcal{G}$ has a (constant-valued) utility function u_g where $u_g \geq 0$, representing how much g is worth to a user; and each action $a \in A$ has an associated execution cost $c_a \geq 0$, representing how costly it is to execute each action (e.g., representing the amount of time or resources consumed). All goals become *soft constraints* so that any plan achieving a subset of goals (even the empty set) is a valid plan. Let $\mathbb{P}$ be the set of all valid plans and let $G_P \subseteq \mathcal{G}$ be the set of goals achieved by a plan $P \in \mathbb{P}$. The objective is to find a plan P that maximizes the difference between total achieved utility $u(G_P)$ and total cost of all actions in P:

$$\arg\max_{P \in \mathbb{P}} \quad \sum_{g \in G_P} u_g - \sum_{a \in P} c_a \tag{2.1}$$

In this chapter, we discuss extensions to this model that provide for goal utility dependencies, or reward dependencies between goals such that achieving a set of goals may have a reward greater (or less) than the sum of each individual goals' reward. After, we define goal costs in the context of temporal planning, where actions have duration and goal achievement after a deadline incurs a penalty cost.

2.1 Goal Utility Dependencies

In partial satisfaction planning (PSP) the process of finding goals on which to focus is complicated by the fact that they interact with one another. For instance, actions may share in their achievement of goals (positive interaction) or conflict (negative interaction). These types of interactions introduce cost dependencies between goals

because the cost of achieving them separately may differ from the cost of achieving them together. In the previously defined goal reward model of PSP *net benefit*, goals only interact through cost dependencies. This work extends PSP to handle utility dependencies. This allows users to specify changes in utility based on the achieved set of goals.

With no utility dependencies on goals their utilities are additive: $u(G_P) = \sum_{g \in G_P} u_g$, where u_g represents the utility of a goal g. To represent goal utility dependencies, we adopt the *Generalized Additive Independence* (GAI) model [2]. This model was chosen because it is expressive, general and we can compile to it from other commonly used models such as UCP-Networks [14]. It defines the utility of the goal set G as *k* local utility functions $f^u(G_k) \in \mathbb{R}$ over sets $G_k \subseteq \mathcal{G}$. For any subset $G' \subseteq \mathcal{G}$ the utility of G' is:

$$u(G') = \sum_{G_k \subseteq G'} f^u(G_k) \tag{2.2}$$

This model allows users to specify changes in utility over sets of goals. We name the new PSP problem with utility dependencies represented by the GAI model ***PSP***$^{\mathcal{UD}}$. If there are $|G|$ local functions $f^k(G_k)$ and each G_k contains a single goal then ***PSP***$^{\mathcal{UD}}$ reduces to the original PSP problem (no utility dependencies).

2.2 Time-dependent Goal Costs

So far we have discussed goal utility functions that are independent of achievement time. That is, the achieved reward is always the same given the same set of achieved goals. But often penalty can be incurred based on *when* a goal is achieved [55]. For example, consider a delivery truck that must deliver goods by a particular deadline and being late means reduced payment. This is a *time-dependent* goal because final value of a goal varies with its achievement time.

Before diving into how these goals are defined, it is important to define temporal actions in temporal planning problems. Temporal planning problems are typically defined using a PDDL2.1 model of actions and time [42]. In these models, durative actions can be split into instantaneous actions like those in classical planning, where the two parts of an action (a "start" and "end" point) are linked via a defined duration and invariant conditions (i.e., conditions that must hold throughout the duration of the action). Hence, we can define a ground PDDL2.1 temporal action a as having three sets of conditions: $pre_{\vdash}$, conditions that must be true at the start of a durative action; $pre_{\dashv}$, the conditions that must be true at the end of a durative action; $pre_{\leftrightarrow}$, the conditions that must hold during the open interval duration of the action (i.e., all time points between the start and end of the action). Effects of actions can occur at the start or end as well, where $eff_{\vdash}$ are the effects that happen at the start of an action and $eff_{\dashv}$ are the effects that happen at the end of an action. The duration of the action is single value $dur \in \mathbb{R}_{\geq 0}$.[1] Actions can execute concurrently, meaning that actions may start before others have finished. It is important to note that starting an action forces its end. That is, the end effects of all actions in a plan must occur before the plan reaches its final goal state. Otherwise, the goal of planning is the same. From an initial state I, a final goal state must be found where all goals in the goal set G are true.

For time-dependent goal costs, we look toward defining a cost function over goal achievement within the temporal planning framework. The idea was first explored by Haddawy and Hanks in the context of planning for optimal utility plans [55]. One can view these as deadline goals, where no penalty cost is given if

[1] In PDDL2.1 actions can include a calculable minimum and maximum duration of an action, but for the sake of simplifying matters, and in all of the domains on which we discuss, we assume that each action has a single, pre-defined duration.

the goal is achieved before a deadline, but afterwards there is a linear increase in cost given for goal achievement until reaching a maximum cost value (at another given time point). We would like to find the lowest cost plan in the presence of such goals.[2]

We model time-dependent goal cost as a function of the goal g and its final achievement time t_g:[3]

$$c(g, t_g) = \begin{cases} 0 & \text{if } t_g \leq t_1 \\ \frac{t_g - t_d}{t_{d+\delta} - t_d} \cdot c_g & \text{if } t_d < t_g \leq t_{d+\delta} \\ c_g & \text{if } t_{d+\delta} < t_g \end{cases}$$

where c_g is the full cost for g, t_d is the soft deadline time for the goal and $t_{d+\delta}$ is the time point where full penalty cost is given for the goal. This function ensures that no cost is given if the goal is achieved before t_d, partial penalty is given if the goal is achieved between t_d and $t_{d+\delta}$ and the full cost is paid if the goal is achieved after $t_{d+\delta}$. For each goal, we sum the costs of their achievement and the problem is to minimize the cost.

[2]This objective is compilable directly into *net benefit* as defined earlier.

[3]We assume a goal can be achieved once (and not deleted then re-achieved). This assumption can hold without loss of generality via the use of compilation techniques to force a dummy goal to become true at the original goal's first or last achievement time.

Chapter 3

Heuristic Search for Maximizing Net Benefit

Effective handling of PSP problems poses several challenges, including an added emphasis differentiating between feasible and "good" plans. Indeed, in classes of PSP problems that involve all soft goals and constraints a trivially feasible, but decidedly non-optimal solution would be the "null" plan; that is, choosing to do nothing and ignoring the goals. In the case of PSP, one has the coupled problem of deciding what goals to pursue (in the case of soft goals), when to achieve them (in the case of time-dependent costs) and finding the best plan to achieve those goals so that we may find the best solution. Choosing goals is further complicated in the presence of goal utility dependencies, were we have to consider both action interactions and goal interactions.

All of the main planning methods in this dissertation have their basis in heuristic search (even the techniques inspired by local search that are discussed in Section 4.4). In this chapter, we discuss the search method used for partial satisfaction planning when maximizing *net benefit*.[1]

3.1 Best-First Heuristic Search for PSP

The planner *Sapa*$^{\mathcal{PS}}$ [7] provides the underlying search algorithm for most of the planners discussed in this dissertation. This best-first, heuristic forward search planner uses an anytime variation of the A^* [56] algorithm guided by a heuristic derived from the relaxed planning graph [63]. Like A^*, this algorithm starts with the initial state S_{init} and continues to dequeue from the open-list the most promising node S

[1]The planner OPTIC, which is used for handling soft temporal planning deadlines and is discussed in Chapter 6 also uses heuristic search. However, its search works toward minimizing penalty costs and uses a search strategy geared toward scaling up for temporal planning.

(i.e., highest $f(s) = g(s) + h(s)$ value). For each search node s, $g(s)$ represents the benefit achieved so far by visiting s from s_{init} and $h(s)$ represents the projected maximum additional benefit gained by expanding s, with plan benefit defined in Section 2.1. Though calculating $g(s)$ is trivial, having a good estimate of $h(s)$ is hard and key to the success of best-first search algorithms. During exploration of the search tree the algorithm keeps outputting better quality plans whenever a node S with the best-so-far $g(s)$ value is expanded (i.e., it outputs a "best" plan upon generating it rather than when a state is expanded). Like A^*, the algorithm terminates when it chooses a node s with $h(s) = 0$ from the open list.

On top of this, the algorithm additionally uses a rudimentary lookahead technique derived from the relaxed plan graph-based heuristic, similar to what is done in the planner YAHSP2 [97], but using a relaxed plan structure and without a repair strategy. Specifically, it takes relaxed plans found during the calculation of the heuristic and repeatedly attempts to simulate their execution until either all actions have been simulated or no further simulation is possible. The resulting state is then added to the search queue, effectively probing deeper into the search space.

In practice, the search algorithm prunes the search space by removing nodes that appear *unpromising* (i.e., nodes where the estimated benefit is negative). Though this improves efficiency, one potential drawback is that when an inadmissible heuristic $h(s)$ underestimates the value of a search node s, then s will be discarded (when compared to the benefit of the best solution found so far $g(s_B)$ from a state s_B) even if it can be extended to reach a better solution. A similar strategy is used in the planner OPTIC, which we use for time-dependent costs, though it always uses an admissible heuristic to prune (and hence does not suffer from this drawback). For the other planners, one difference from *Sapa*$^{\mathcal{PS}}$, is that the algorithm is modified

to keep some search nodes that appear unpromising when first generated. During search it sets a value ϵ as half the distance between the best node found so far s_B and the worst-valued unpromising node. For each unpromising search node s that is within a threshold ϵ of the current best solution, it finds ρ, the complement of the percentage distance between it and the benefit of s_B (i.e., $g(s_B)$). It then keeps s with probability ρ. Note that it only uses this method when applying inadmissible heuristics.

Anytime Best-First Search Algorithm for PSP: One of the most popular methods for solving planning problems is to cast them as the problem of searching for a minimum cost path in a graph, then use a heuristic search to find a solution. Many of the most successful heuristic planners [13, 63, 31, 79, 91] employ this approach and use variations of best-first graph search (BFS) algorithms to find plans. We also use this approach to solve PSP *net benefit* problems. In particular, many of the planners in this dissertation use a variation of A^* with modifications to handle some special properties of PSP *net benefit* (e.g., any state can be a goal state when all goals are soft). The remainder of this section will outline them and discuss the search algorithm in detail.

Standard shortest-path graph search algorithms search for a minimum-cost path from a start node to a goal node. Forward state space search for solving classical planning problems can be cast as a graph search problem as follows: (1) each search node n represents a complete planning state s; (2) if applying action a to a state s leads to another state s' then action a represents a directed edge $e = s \xrightarrow{a} s'$ from s to s' with the edge cost $c_e = c_a$; (3) the start node represents the initial state I; (4) a goal node is any state s_G satisfying all goals $g \in G$. In our ongoing example, at the initial state $I = \{at(A)\}$, there are four applicable actions $a_1 = Move(A, B)$, $a_2 =$

$Move(A, C)$, $a_3 = Move(A, D)$, and $a_4 = Move(A, E)$ that lead to four states $s_1 = \{at(B), g_1\}$, $s_2 = \{at(C), g_2\}$, $s_3 = \{at(D), g_3\}$, and $s_4 = \{at(E), g_4\}$. The edge costs will represent action costs in this planning state-transition graph[2] and the shortest path in this graph represents the lowest cost plan. Compared to the classical planning problem, the PSP *net benefit* problem differs in the following ways:

- Not all goals need to be accomplished in the final plan. In the general case where all goals are *soft*, any executable sequence of actions is a candidate plan (i.e., any node can be a valid goal node).
- Goals are not uniform and have different utility values. The plan quality is not measured by the total action cost but by the difference between the cumulative utility of the goals achieved and the cumulative cost of the actions used. Thus, the objective function shifts from *minimizing* total action cost to *maximizing* net benefit.

To cast PSP *net benefit* as a graph search problem, some changes are necessary so that (1) the edge weight representing the change in plan benefit by going from a search node to its successors and (2) the criteria for terminating the search process coincides with the objective of maximizing net benefit. First comes a discussion on the modifications, then a discussion on a variation of the A^* search algorithm for solving the graph search problem for PSP. To simplify the discussion and to facilitate proofs of certain properties of this algorithm, the algorithm will make the following assumptions: (1) all goals are soft constraints; (2) the heuristic is admissible. Later follows a discussion about relaxing one or more of those assumptions.

[2]In the simplest case where actions have no cost and the objective function is to minimize the number of actions in the plan, the algorithm can consider all actions having uniform positive cost.

g-value: A^* uses the value $f(s) = g(s) + h(s)$ to rank generated states s for expansion with g representing the "value" of the (known) path leading from the start state I to s, and h estimating the (unknown) path leading from s to a goal node that will optimize a given objective function. In PSP *net benefit*, g represents the additional benefit gained by traveling the path from I to s. For a given state s, let $G_s \subseteq G$ be the set of goals accomplished in s, then:

$$g(s) = (U(s) - U(I)) - C(P_{I \to s}) \tag{3.1}$$

where $U(s) = \sum_{g \in G_s} u_g$ and $U(I) = \sum_{g \in G_I} u_g$ are the total utility of goals satisfied in s and I. $C(P_{I \to s}) = \sum_{a \in P_{I \to s}} c_a$ is the total cost of actions in $P_{I \to s}$. For example: $U(s_2) = u_{g_2} = 100$, and $C(P_{I \to s_2}) = c_{a_2} = 90$ and thus $g(s_2) = 100 - 90 = 10$.

In other words, $g(s)$ as defined in Equation 3.1 represents the additional benefit gained when plan $P_{I \to s}$ is executed in I to reach s. To facilitate the discussion, we use a new notation to represent the benefit of a plan P leading from a state s to another state s':

$$B(P|s) = (U(s') - U(s)) - \sum_{a \in P} c_a \tag{3.2}$$

Thus, we have $g(s) = B(P_{I \to s}|I)$.

h value: In graph search, the heuristic value $h(s)$ estimates the path from s to the "best" goal node. In PSP *net benefit*, the "best" goal node is the node s_g such that traveling from s to s_g will give the most additional benefit. In general, the closer that h estimates the real optimal h^* value, the better in terms of the amount of search effort. Therefore, we first introduce the definition of h^*.

Best beneficial plan: *For a given state s, a best beneficial plan P_s^B is a plan executable in s and there is no other plan P executable in s such that: $B(P|s) > B(P_s^B|s)$.*

Notice that an empty plan $P_\emptyset$ containing no actions is applicable in all states and $B(P_\emptyset|s) = 0$. Therefore, $B(P_s^B|s) \geq 0$ for any state s. The optimal additional achievable benefit of a given state s is calculated as follows:

$$h^*(s) = B(P_s^B|s) \tag{3.3}$$

In our ongoing example, from state s_2, the most beneficial plan is $P_{s_2}^B = \{Move(C, D), Move(D, E)\}$, and $h^*(s_2) = B(P_{s_2}^B|s_2) = U(\{g_3, g_2, g_4\}) - U(\{g_2\}) - (c_{Move(C,D)} + c_{Move(D,E)}) = ((300 + 100 + 100) - 100) - (200 + 50) = 400 - 250 = 150$. Computing h^* directly is impractical as the algorithm needs to search for P_s^B in the space of all potential plans and this is as hard as solving the PSP *net benefit* problem for the current search state. Therefore, a good approximation of h^* is needed to effectively guide the heuristic search algorithm.

Figure 3.1 describes the anytime variation of the A^* algorithm that is used to solve the PSP *net benefit* problems. Like A^*, this algorithm uses the value $f = g + h$ to rank nodes to expand, with the successor generator and the g and h values described above. It is assumed that the heuristic used is *admissible*. Because the algorithm tries to find a plan that maximizes *net benefit*, admissibility means overestimating additional achievable benefit; thus, $h(s) \geq h^*(s)$ with $h^*(s)$ defined above. Like other anytime algorithms, the algorithm keeps one incumbent value

SEARCH($\langle F, I, G, A \rangle$)

1. $g(I) \leftarrow \sum_{g \in I} u_g$
2. $f(I) \leftarrow g(I) + h(I)$
3. $B_B \leftarrow g(I)$
4. $P_B \leftarrow \emptyset$
5. $OPEN \leftarrow \{I\}$
6. while $OPEN \neq \emptyset$ and not *interrupted* do
7. $\quad s \leftarrow \arg\max_{x \in OPEN} f(x)$
8. $\quad OPEN \leftarrow OPEN \setminus \{s\}$
9. $\quad$ if $h(s) = 0$
10. $\quad\quad$ stop search
11. $\quad$ else
12. $\quad\quad$ foreach $s' \in Successors(s)$
13. $\quad\quad\quad$ if $g(s') > B_B$
14. $\quad\quad\quad\quad P_B \leftarrow$ plan leading from I to s'
15. $\quad\quad\quad\quad B_B \leftarrow g(s')$
16. $\quad\quad\quad\quad OPEN \leftarrow OPEN \setminus \{s_i : \ f(s_i) \leq B_B\}$
17. $\quad\quad\quad$ if $f(s') > B_B$
18. $\quad\quad\quad\quad OPEN \leftarrow OPEN \cup \{s'\}$
19. Return P_B

Figure 3.1: Anytime A* search algorithm.

B_B to indicate the quality of the best found solution at any given moment (i.e., highest net benefit).[3]

The search algorithm starts with the initial state I and keeps expanding the most promising node s (i.e., one with highest f value) picked from the *OPEN* list. If $h(s) = 0$ (i.e., the heuristic estimate indicates that there is no additional benefit gained by expanding s) the algorithm stops the search. This is true for the termination criteria of the A^* algorithm (i.e., where the goal node gives $h(s) = 0$). If $h(s) > 0$, then it expands s by applying applicable actions a to s to generate all

[3]Figure 3.1, as implemented in our planners is based on *Sapa*$^{\mathcal{PS}}$ and does not include duplicate detection (i.e., no *CLOSED* list). However, it is quite straightforward to add duplicate detection to the base algorithm similar to the way *CLOSED* list is used in A^*.

successors.[4] If the newly generated node s' has a better $g(s')$ value than the best node visited so far (i.e., $g(s') > B_B$), then it records $P_{s'}$ leading to s' as the new best found plan. Finally, if $f(s') \leq B_B$ (i.e., the heuristic estimate indicates that expanding s' will never achieve as much additional benefit to improve the current best found solution), it will discard s' from future consideration. Otherwise s' is added to the $OPEN$ list. Whenever a better solution is found (i.e., the value of B_B increases), it will also remove all nodes $s_i \in OPEN$ such that $f(s_i) \leq B_B$. When the algorithm is interrupted (either by reaching the time or memory limit) before the node with $h(s) = 0$ is expanded, it will return the best plan P_B recorded so far (the alternative approach is to return a new best plan P_B whenever the best benefit value B_B is improved). Thus, compared to A^*, this variation is an "anytime" algorithm and always returns some solution plan regardless of the time or memory limit.

Like any search algorithm, one desired property is preserving optimality. If the heuristic is admissible, then the algorithm will find an optimal solution if given enough time and memory.[5]

Proposition 1: *If h is admissible and bounded, then the algorithm in Figure 3.1 always terminates and the returned solution is optimal.*

Proof: Given that all actions a have constant cost $c_a > 0$, there is a finite number of sequences of actions (plans) P such that $\sum_{a \in P} c_a \leq U_G$. Any state s generated by

[4]Note that with the assumption of $h(s)$ being admissible, we have $h(s) \geq 0$ because it overestimates $B(P_s^B|s) \geq 0$.

[5]Given that there are both positive and negative edge benefits in the state transition graph, it is desirable to show that there is no positive cycle (any plan involving positive cycles will have infinite achievable benefit value). Positive cycles do not exist in our state transition graph because traversing over any cycle does not achieve any additional utility but always incurs positive cost. This is because the utility of a search node s is calculated based on the world state encoded in s (not what accumulated along the plan trajectory leading to s), which does not change when going through a cycle c. However, the total cost of visiting s is calculated based on the sum of action costs of the plan trajectory leading to s, which increases when traversing c. Therefore, all cycles have non-positive net benefit (utility/cost trade-off).

plan P such that $\sum_{a \in P} c_a > 2 \times U_G$ will be discarded and will not be put in the *OPEN* list because $f(s) < 0 \leq B_B$. Given that there is a finite number of states that can be generated and put in the *OPEN* list, the algorithm will exhaust the *OPEN* list given enough time. Thus, it will terminate.

The algorithm in Figure 3.1 terminates when either the *OPEN* list is empty or a node s with $h(s) = 0$ is picked from the *OPEN* list for expansion. First we see that if the algorithm terminates when $OPEN = \emptyset$, then the plan returned is the optimal solution. If $f(s)$ overestimates the real maximum achievable benefit, then the discarded nodes s due to the cutoff comparison $f(s) \leq B_B$ cannot lead to nodes with higher benefit value than the current best found solution represented by B_B. Therefore, our algorithm does not discard any node that can lead to an optimal solution. For any node s that is picked from the *OPEN* list for expansion, we also have $g(s) \leq B_B$ because B_B always represents the highest g value of all nodes that have ever been generated. Combining the fact that no expanded node represents a better solution than the latest B_B with the fact that no node that was discarded from expansion (i.e., not put in or filtered out from the *OPEN* list) may lead to a better solution than B_B, we can conclude that if the algorithm terminates with an empty *OPEN* list then the final B_B value represents the optimal solution.

If the algorithm in Figure 3.1 does not terminate when $OPEN = \emptyset$, then it terminates when a node s with $h(s) = 0$ was picked from the *OPEN* list. We can show that s represents the optimal solution and the plan leading to s was the last one output by the algorithm. When s with $h(s) = 0$ is picked from the *OPEN* list, given that $\forall s' \in OPEN : f(s) = g(s) \geq f(s')$, all nodes in the *OPEN* list cannot lead to a solution with higher benefit value than $g(s)$. Moreover, let s_B represent the state for which the plan leading to s_B was last output by the algorithm; thus $B_B = g(s_B)$.

If s_B was generated before s, then because $f(s) = g(s) < g(s_B)$, s should have been discarded and was not added to the *OPEN* list, which is a contradiction. If s_B was generated after s, then because $g(s_B) \geq g(s) = f(s)$, s should have been discarded from the *OPEN* list when s_B was added to the *OPEN* list and thus s should not have been picked for expansion. Given that s was not discarded, we have $s = s_B$ and thus P_s represents the last solution output by the algorithm. As shown above, none of the discarded nodes or nodes still in the *OPEN* list when s is picked can lead to better solution than s, where s represents the optimal solution. □

Discussion: Proposition 1 assumes that the heuristic estimate h is bounded and this can always be done. For any given state s, Equation 3.3 indicates that $h^*(s) = B(P_s^B|s) = (U(s') - U(s)) - \sum_{a \in P_s^B} c_a \leq U(s') = \sum_{g \in s'} u_g \leq \sum_{g \in G} u_g = U_G$. Therefore, it is possible to safely assume that any heuristic estimate can be bounded so that $\forall s : h(s) \leq U_G$.

To simplify the discussion of the search algorithm described above, several assumptions were made at the beginning of this section: all goals are soft, the heuristic used is admissible, the planner is forward state space, and there are no constraints beyond classical planning. If any of those assumptions is violated, then some adjustments to the main search algorithm are necessary or beneficial. First, if some goals are "hard goals", then only nodes satisfying all hard goals can be termination nodes. Therefore, the condition for outputting the new best found plan needs to be changed from $g(s') > B_B$ to $(g(s') > B_B) \wedge (G_h \in s)$ where G_h is the set of all hard goals.

Second, if the heuristic is inadmissible, then the final solution is not guaranteed to be optimal. To preserve optimality, it is possible to place all generated nodes in

the OPEN list. Finally, if there are constraints beyond classical planning such as metric resources or temporal constraints, then adjustments must be made to the state representation. Indeed, in the case of temporal problems, other search algorithms may be more suitable so that temporally expressive planning problems can be handled [27]. To these ends, Chapter 6 discusses the use of a different baseline planner that is suitable for dealing with temporally expressive planning problems [24, 23] for soft temporal deadlines.

Chapter 4

Solving for Goal Utility Dependencies

While solving for goals that have individual rewards offers its own set of challenges, handling goal utility dependencies presents its own issues. If dependencies are defined such that only positive reward is given for achieving a set of goals, then we have the same problem as having individual rewards (i.e., for every goal set we can define a dummy goal with reward that becomes true when the set becomes true). However, with negative rewards the situation becomes more difficult in practice. Indeed, heuristics based on ignoring delete lists of actions have difficulty picking up on negative penalties. That is, when a goal independently looks beneficial but gives a negative value when combined with other goals, simply generating dummy sets will not work. The heuristic will assume the "cheapest path" to each goal set, effectively making the assumption that only the positive benefits of goal achievement. The issue is that these heuristics typically only consider the cheapest cost of goal reachability, ignoring decisions on whether to achieve particular sets of end goals based on negative rewards.

This chapter discusses methods to handle problems with goal utility dependencies. It first briefly discusses a technique that can extend certain integer program (IP) encodings of planning problems to include constraints on goal utility dependencies. The main disadvantage of this approach is that IP encodings of problems require a limit on the plan length (i.e., it limits the planning horizon such that optimality can never be fully guaranteed), and therefore are only optimal to some bound. Hence, we cover heuristics that combine planning graph methods with a declarative integer program (IP) encoding. The first heuristics generate an IP en-

coding over the relaxed plan heuristic. In these heuristics, the IP encoding selects a goal set along with an estimated cost for achieving it. With this method it is possible to generate admissible and inadmissible heuristics, where the admissible heuristic can guarantee optimal solutions when the search algorithm terminates. The main innovation is the combination of a relaxed plan that handles cost interactions between goals and a declarative IP encoding that captures both mutual goal achievement cost and goal utility dependencies. We then introduce and discuss an IP-based admissible heuristic that relies on an action ordering relaxation, which then is further relaxed to a linear program (LP). And finally, we discuss a learning method that can be used to improve plan quality in some cases.

4.1 IP ENCODING FOR $PSP^{\mathcal{UD}}$

Since classical planning problems can be solved by IP, and since IP provides a natural way to incorporate numeric constraints and objective functions, it follows that $PSP^{\mathcal{UD}}$ planning problems can be solved by IP as well.

This section discusses an IP formulation to handle $PSP^{\mathcal{UD}}$ problems by extending the generalized single state change (G1SC) formulation [96]. Currently, the G1SC formulation is the most effective IP formulation for solving classical planning problems, and it outperforms the previously developed IP formulation used to solve PSP problems without utility dependencies [95].

The G1SC formulation represents the planning problem as a set of loosely coupled network flow problems, where each network corresponds to one of the state variables in the planning domain. The network nodes correspond to the state variable values and the network arcs correspond to the value transitions. The planning problem is to find a path (a sequence of actions) in each network such that, when merged, they constitute a feasible plan. In the networks, nodes and arcs appear in

layers, where each layer represents a plan period. The layers are used to solve the planning problem incrementally. That is, we start by performing reachability analysis to find a lower bound on the number of layers necessary to solve the problem. If no plan is found, all the networks are extended by one extra layer and the planning problem is solved again. This process is repeated until a plan is found (see [96] for a complete description of the G1SC formulation).

In order to deal with utility dependencies we incorporate four extensions to the G1SC formulation:

- In *PSP*$^{\mathcal{UD}}$ problems, not all goals have to be achieved for a plan to be feasible. Therefore, we remove those constraints from the G1SC formulation which state that goals must be achieved.

- For each goal utility dependency function G_k, we add a variable $z_{G_k} \in \{0, 1\}$, where $z_{G_k} = 1$ if all goals in G_k are achieved, and $z_{G_k} = 0$ otherwise.

- For each goal utility dependency function G_k, we add constraints to ensure that G_k is satisfied if and only if all goals $g \in G_k$ are achieved, that is:

$$\sum_{f,g \in D_c : g \in G_k} y_{c,f,g,T} - |G_k| + 1 \leq z_{G_k} \tag{4.1}$$

$$z_{G_k} \leq \sum_{f \in D_c} y_{c,f,g,T} \quad \forall g \in D_c : g \in G_k \tag{4.2}$$

  where D_c is the domain of a state variable c, $y_{c,f,g,T} \in \{0, 1\}$ are variables of the IP problem that represent value changes in the state variables, and T is the plan horizon.

- We create an objective function to maximize the net-benefit (utility minus cost) of the plan.

$$\text{MAX} \quad \sum_{G_k} u(G_k) z_{G_k} - \sum_{a \in A, 1 \leq t \leq T} c_a x_{a,t} \tag{4.3}$$

where $u(G_k)$ represents the utility of satisfying the goal utility dependency function G_k, and c_a represents the cost of executing action $a \in A$.

The extended G1SC formulation is bounded length optimal (i.e., it generates optimal plans for a plan horizon T). Global optimality cannot be guaranteed as there could still be solutions with higher net benefit at longer plan horizons.

4.2 Delete Relaxation Heuristics for Goal Utility Dependencies

A relaxed planning graph is created by iteratively applying all possible applicable actions given the propositions available, thereby generating a union of the previously available propositions with the ones added by applying the actions. This can provide a cost estimate on reaching a particular proposition by summing the cost of each action applied to reach it, always keeping the minimum summed cost (i.e., the cheapest cost to reach any proposition). This process is called *cost propagation*. After this, we can extract a relaxed plan from the planning graph by finding the supporting actions for the set of goals. The heuristic value is typically taken from the sum of the cost of all actions in the relaxed plan. If we could extract an optimal relaxed plan the heuristic would be admissible. However, due to the difficulty of this task (which is NP-hard [19]) greedier approaches are generally used (such as preference for the cheapest supporting action at each step).

In these heuristic methods we estimate the cost $\mathcal{C}(g)$ to achieve each goal [33]. Starting with $\mathcal{C}(f) = 0$ for facts f in the initial state I and $\mathcal{C}(f) = \mathcal{C}(a) = \infty$ for all other facts and all actions, the propagation rules to estimate costs to achieve facts p and to execute actions a are:[1]

[1] c_a, which is the execution cost of a, is different from $\mathcal{C}(a)$, which is the estimated cost to enable the execution of a (i.e., costs to achieve preconditions of a)

- Facts: $\forall f : \mathcal{C}(f) = \underset{f \in Add(a)}{MIN} (\mathcal{C}(a) + c_a)$

    1. Max-prop: $\forall a \in A : \mathcal{C}(a) = \underset{f \in Pre(a)}{MAX} \mathcal{C}(f)$; or
    2. Sum-prop: $\forall a \in A : \mathcal{C}(a) = \underset{f \in Pre(a)}{\Sigma} \mathcal{C}(f)$

The update rules are used while extending a (relaxed) planning graph structure [11]. After the propagation is done (i.e., no costs change), $\mathcal{C}(g)$ is an estimate on the cost to achieve g for each goal $g \in \mathcal{G}$.

Deriving Heuristics from Propagated Costs

This dissertation will use the notation h^x_y to name the heuristics. Here x is the method used to define the goal utilities and y is the method used to estimate the goal costs. The dependencies between goal utilities can be defined using the GAI model (discussed in Chapter 2) while the dependencies between goal costs can be estimated using relaxed plans.[2]

It is easy enough to observe that if we use *max* propagation (max-prop), then $\mathcal{C}(g)$ will underestimate the cost to achieve g while there is no such guarantee for *sum* propagation (sum-prop) [13]. With max propagation, we have an admissible heuristic, allowing optimal solutions to be found. Using $\mathcal{C}(g)$ calculated by the cost propagation process outlined, we can estimate the achievable benefit value as:

$$h^{GAI} = \underset{G' \subseteq G}{MAX} \ [u(G') - (\underset{g \in G'}{MAX} \mathcal{C}(g))] \tag{4.4}$$

Notice part of the heuristic includes the local utility functions as defined in Equation 2.2 (see Section 2.1). As such, the heuristic directly applies the GAI model. If using max-prop, then Equation 4.4 will give the h^{GAI}_{max} heuristic and if

[2]Given this notation, we can view the heuristic used in the planner *Sapa*$^{\mathcal{PS}}$ [7] as h^{sum}_{relax} because it sums the individual goal utilities and extracts a relaxed plan to estimate cost.

using sum-prop, it will give a corresponding h^{GAI}_{sum} heuristic. While h^{GAI}_{max} overestimates the real achievable benefit, there is no such guarantee for h^{GAI}_{sum}. Recall that since the problem involves maximizing net benefit, an heuristic that always overestimates is required to maintain admissibility. The admissibility of h^{GAI}_{max} is maintained since the goal utility dependencies are solved for directly (with the cost estimates from *max* propagation) or in a relaxed fashion. In other words, since *max* propagation provides an underestimate of individual costs, and h^{GAI}_{max} solves the goal utility dependencies exactly, its admissibility is maintained since the heuristic will always provide an overestimate of total achievable *net benefit*.

To handle the goal utility dependencies with the propagated cost, the heuristic solves the following integer program to get the final heuristic value, where C represents the propagated cost value:

- Binary Variables:
  - $\forall g \in G, \forall G_k \subseteq G,\ f^u(G_k) \neq 0$: create one binary integer variable X_g, X_{G_k}.
- Constraints:
  - $\sum_{g \in G_k} (1 - X_g) + X_{G_k} \geq 1$
  - $\forall g \in G_k :\ (1 - X_{G_k}) + X_g \geq 1$
- Objective: MAX $(\sum f^u(G_k) * X_{G_k} - C)$.

Relaxed Plan-based Heuristic

h^{GAI}_{max} can easily offer a high overestimate on the *net benefit*, since it relies on max propagation, a weak estimate on the cost to achieve individual goals. The h^{GAI}_{sum} heuristic, while more informative, relaxes the cost interaction and assumes that

plans achieving different goals are independent and do not overlap. To improve on this, it is possible to adapt the relaxed plan heuristic, first introduced in the FF planner [63], that solves a relaxation of the planning problem by delete effects (also called the "delete list"). This heuristic offers improvements over h^{GAI}_{sum} by taking into account actions contributing to the achievement of several goals. The challenge in extending it to PSP with goal utility dependencies is how to efficiently find a high-benefit relaxed plan in the presence of both cost and utility dependencies.

Let $G_{P^+} \subseteq \mathcal{G}$ be the set of goals achieved by the relaxed plan P^+. The relaxed plan heuristic for ***PSP***$^{\mathcal{UD}}$ is:

$$h^{*\ GAI}_{relax} = \underset{P^+}{\text{MAX}} \quad u(G_{P^+}) - \sum_{a \in P^+} c_a \tag{4.5}$$

Note that Equation 4.5 looks like Equation 2.1 except that the optimal plan P in Equation 2.1 is replaced by the optimal relaxed plan P^+ (i.e., one achieving maximum benefit for the relaxed problem) in Equation 4.5. $h^{*\ GAI}_{relax}$ overestimates the real achievable benefit and can be used as an admissible heuristic in the search to find the optimal solution for ***PSP***$^{\mathcal{UD}}$ problems.

While finding a satisfying relaxed plan P^+ for any given goal set $G_{P^+} \subseteq \mathcal{G}$ is polynomial, extracting $h^{*\ GAI}_{relax}$ requires finding an optimal relaxed plan (highest benefit). This task is NP-hard even when we already know the optimal goal set $G^*_{P^+}$ and actions have uniform cost [19]. To approximate $h^{*\ GAI}_{relax}$ for ***PSP***$^{\mathcal{UD}}$ the heuristic uses the following three steps. The first two steps were introduced in the planner ***Sapa***$^{\mathcal{PS}}$ while the third step is novel:

1. Greedily extract a low cost relaxed plan P^+ that achieves the *largest* set of achievable goals.

2. Capture the achievement cost dependencies between achievable goals using the causal structure of P^+.

3. Pose the problem of extracting the optimal subplan within P^+ that takes both cost and utility dependencies into account as an IP encoding. A solution h^{GAI}_{relax} of this IP encoding is used to estimate $h^{*\ GAI}_{relax}$.

Step 1: Heuristically Extract a Low Cost Relaxed Plan: Let $G' \subseteq \mathcal{G}$ be the set of all achievable goals ($\mathcal{C}(g) < \infty$). The heuristic uses the planning graph and the propagated achievement costs to heuristically extract a low-cost relaxed plan to support G' as follows:

1. Start with supported facts $SF = I$, subgoal set $SG = G' \setminus I$ and the relaxed plan $P^+ = \emptyset$.

2. For each $g \in SG$ select a supporting action $a : g \in Add(a)$ with lowest execution cost $\mathcal{C}(a)$ value. Update: $P^+ \leftarrow P^+ \cup \{a\}$, $SG \leftarrow SG \cup (Pre(a) \setminus SF)$ and $SF \leftarrow SF \cup Add(a)$.

3. Repeat until $SG = \emptyset$.

This backtrack-free process is guaranteed to finish in time polynomial in the number of actions.

Step 2: Build Cost Dependencies within P^+: Because certain actions contribute to the achievement of multiple goals, there are dependencies between the costs to achieve them. Those relations can be discovered by using the causal structure of the extracted relaxed plan P^+.

To capture the mutual dependencies between the goal achievement costs, the heuristic finds the set of actions shared between different partial plans achieving different goals. This uses the causal links in the relaxed plan P^+.

$$GS(a) = \bigcup_{p \in Effect(a)} GS(p) \tag{4.6}$$

$$GS(p) = \begin{cases} p \cup (\bigcup_{p \in Prec(a)} GS(a)) & \text{if } p \in G \\ \bigcup_{p \in Prec(a)} GS(a) & \text{if } p \notin G \end{cases} \tag{4.7}$$

Using the above equations for each action a, $GS(a)$ contains the set of goals g that a contributes to, where the goal-supporting sets $GS(a)$ represent the achievement cost dependencies between goals.

Step 3: Estimate the Maximum Achievable Benefit: In this step, the heuristic combines the goal supporting set $GS(a)$ found in the previous step with the goal utility dependencies f^u to find the most beneficial relaxed plan P' within P^+. One naive approach to find $P' \subseteq P^+$ is to iterate over all $2^{|G_{P^+}|}$ subsets $G' \subseteq G_{P^+}$ of goals, where G_{P^+} is the set of goals achieved by P^+, and compare the benefit of plans P' achieving G'. However, when $|G|$ is large this approach becomes impractical. Therefore, the heuristic uses a declarative approach of setting up an IP encoding with its solution representing the most beneficial relaxed plan $P' \subseteq P^+$. Note that while IP is generally slow, the number of actions in the relaxed plan is much smaller an IP encoding of the entire (relaxed) planning graph, giving a relatively reasonable heuristic solving time per node. The heuristic's IP has constraints representing the goal supporting set $GS(a)$ found in the previous step. These enforce the fact that if a given goal g is selected, then any action that contributes to the achievement of g should also be selected. The final heuristic IP encoding looks

very similar to that used for h^{GAI}_{max} and h^{GAI}_{sum}, with added constraints on the actions. Specifically:

- Binary Variables:
  - $\forall a \in P, \forall g \in G, \forall G_k \subseteq G,\ f^u(G_k) \neq 0$: create one binary integer variable X_a, X_g, X_{G_k}.
- Constraints:
  - $\forall a \in P, \forall g \in GS(a) : (1 - X_g) + X_a \geq 1$
  - $\sum_{g \in G_k} (1 - X_g) + X_{G_k} \geq 1$
  - $\forall g \in G_k :\ (1 - X_{G_k}) + X_g \geq 1$
- Objective: MAX $(\sum f^u(G_k) * X_{G_k} - \Sigma X_a * c_a)$

Solving this IP encoding gives the benefit value of the most beneficial relaxed plan P' within P^+. The benefit of this P' plan can be used as a h^{GAI}_{relax} heuristic to guide search.

Evaluation

We implemented the heuristic framework on top of the *Sapa*$^{\mathcal{PS}}$ planner [7] and compared it with the discussed IP-based encoding of a bounded-length version of the planning problem. We call the heuristic planner *SPUDS* and IP approach *iPUD*. *SPUDS* is compared using the three heuristics we describe (h^{GAI}_{relax}, h^{GAI}_{max}, and h^{GAI}_{sum}) along with a version of *Sapa*$^{\mathcal{PS}}$ whose heuristic ignores the goal utility dependencies (but whose state evaluation does not).

iPUD runs with CPLEX 10.0, a commercial LP solver, while we use *lp_solve* version 5.5 (a free solver with a Java wrapper) to solve the IP encodings in *SPUDS*.

We found that *lp_solve*, while less powerful than CPLEX, has a shorter IP setup time and is more suitable for *SPUDS*, which sets up an IP encoding at every search node. All tests use a P4 2.66GHz/1GB RAM computer with a 600 second time limit. *SPUDS* and *Sapa*$^{\mathcal{PS}}$ continuously find better solutions until a termination criterion is met.

Test Problems: The *PSP*$^{\mathcal{UD}}$ problems were automatically generated from a subset of the propositional planning benchmarks used in IPC3 and IPC5: In *zenotravel*, airplanes move people between cities; in *satellite*, satellites turn to objects and take pictures; in *rovers*, rovers navigate an area to take samples and images; and in *TPP*, trucks visit markets to buy products.

For each domain, we implemented a Java program that parses the original problem files and generates the *PSP*$^{\mathcal{UD}}$ version with action cost and goal utilities randomly generated within appropriate upper and lower bounds. The set of goal dependencies along with their utility values were also randomly generated. Thus, the number of dependencies, size of the dependencies, set of goals involved, utility values and action costs were all selected within varied lower and upper bounds for each domain. All goals are soft, and therefore planners can trivially solve each problem with the null plan.

For these tests, we varied our bounds on action cost and goal set utility values such that each domain focuses on different aspects of utility dependency. In zenotravel, ending a plan with people at various locations changes utility significantly, and flying a person between locations has a cost that is only slightly less than the individual utilities of achieving each goal. Thus, it is vital to have the certain sets of people at various locations. In TPP, purchasing items has a cost about equiv-

alent to the individual utility of having the item. However, having items together can change the utility of a plan considerably. The idea is to simulate the benefit of having several items together (e.g., to build a crate you need wood, nails, a hammer and saw). The satellite domain removes the emphasis on cost. Here actions have costs lower than the comparatively higher benefit of having several images (e.g., to produce a mosaic image). The domain also adds several negative goal utility dependencies (i.e., substitution) by including negative utility for having certain sets of images yet ending a plan by pointing to an inconvenient spot and having only a few images (e.g., a "partial mosaic"). The rovers domain focuses on substitution as having certain scientific data together can give redundant information and therefore remove a large portion of utility gained by having them separate.

Sapa^{PS} has a heuristic that only takes cost dependencies into account, such that it will remove goals from its heuristic calculation only if the cost of reaching a goal appears greater than its reward. In TPP and zenotravel, the achievement cost for a single goal is about equivalent to or is (more often) greater than the reward obtained for the independent goal reward. Since the *Sapa*^{PS} heuristic looks only at cost dependencies between goals, it is unlikely that it will choose a good (or very large) goal set in these domains. With the rovers and satellite domains, negative goal utility dependencies exist that effectively negate the benefit of simply achieving goals one after the other. That is, it is often the case in those domains that achieving two goals together has reward much less than the independent rewards given for having both goals (such a strategy would yield a negative *net benefit*). This is an especially pronounced feature of the satellite domain. In rovers, the cost of navigating between waypoints where samples may be taken plays a role as well. In the satellite domain, the heuristic of *Sapa*^{PS} is likely to select an (incorrect)

large set of goals, having ignored negative goal utility dependencies, and in the rovers domain, it may select an improper goal set due to goal utility dependencies and action costs.

Analysis: The results in Figure 4.1 show the plan quality achieved by each planning method (top graph) and the time to reach that quality (bottom graph). On problems where only the null plan was found, we indicate the extensive search for a better plan by setting the time to 600 seconds. For every other instance, the time that the best plan was found is shown. As the figure shows, the tested approaches varied in their relative plan quality on each domain but *SPUDS* using the h^{GAI}_{relax} heuristic always performed among the best.

Both the zenotravel and TPP domains involve gathering objects, though zenotravel focuses on delivering these objects as well. Positive utility dependencies play an important role in these domains, since the cost of achieving a single goal often outweighs the individual reward it gives. We see that *Sapa*$^{\mathcal{PS}}$ does poorly, while the *SPUDS* heuristics and *iPUD* fared much better. Since the *Sapa*$^{\mathcal{PS}}$ heuristic is not informed about utility dependencies, this comes as no surprise. In easier problems, the h^{GAI}_{sum} heuristic tends to return plans of similar or equal quality as compared with the other techniques used. However, as problem size increases, h^{GAI}_{sum} begins to return plans of better quality, but still does worse than h^{GAI}_{relax} in terms of the overall number of plans found with best quality. With the IP-only approach, *iPUD*, as the size of the problem increases it is unable to find a good feasible solution.

For our version of the satellite domain, goal combinations remove utility from the overall quality of plans. Also, the plans of higher quality tend to require many actions. This can be seen in the quality of the plans that *iPUD* returns. Its reachability analysis is unable to properly estimate the distance to goals and it therefore

begins its solution searching at a small horizon. For the h^{GAI}_{relax} heuristic, it turns out that action selection helps guide search toward the goals.

For the rovers domain, *iPUD* does well on several problems. However, like in the satellite domain, better quality plans require a larger horizon on some of the problems than its initial horizon provides. This gives *SPUDS* with the h^{GAI}_{relax} heuristic an edge over *iPUD* in 8 of the 20 problems. The heuristics h^{GAI}_{sum} and h^{GAI}_{max} have information regarding utility dependencies, though h^{GAI}_{sum} often performs worse than h^{GAI}_{relax} (solving 5 of 20 problems with better quality plans) and h^{GAI}_{max} is only able to find the null plan in every problem instance for rovers, likely because it cannot detect the cost dependencies between actions in this version of the domain.

Also of interest is the time it takes to solve each problem between the heuristic search methods and the IP encoding used in *iPUD*. Since the *SPUDS* heuristics solve an IP encoding at each search node, they take much longer to compute on larger problems than the procedural *Sapa*$^{\mathcal{PS}}$ heuristic. Unfortunately, *Sapa*$^{\mathcal{PS}}$ lacks the heuristic guidance necessary to properly select goals with utility dependencies. Though we found that the per-node IP encoding of h^{GAI}_{relax} increased the amount of time spent per search node by 3 to 200 times over that of *Sapa*$^{\mathcal{PS}}$ (with the highest increases on larger problems), *SPUDS* with the h^{GAI}_{relax} heuristic does better overall.

When reaching the time limit (600 seconds for our results), *Sapa*$^{\mathcal{PS}}$, *SPUDS* and *iPUD* return their best solution. In *SPUDS* and *Sapa*$^{\mathcal{PS}}$ this behavior comes from the best first anytime search and with *iPUD* this behavior comes from the CPLEX solver, which can return the best feasible solution found within a given time limit. Insights can be obtained by observing the amount of time it takes to find the solution that is eventually returned. We used the anytime behavior to illustrate the

scalability of each approach. Figure 4.2 shows, of problems 10 through 20 in each domain (i.e., the most difficult), which technique performs best in terms of quality throughout their search (e.g., h^{GAI}_{relax} has the best quality for 16 of the problems at 2 seconds). Of our approaches, h^{GAI}_{relax} performs the best overall. In the 80 tested problems, it solves 22 instances at 600 seconds better than any other planner. Also interesting is that in 45 instances it obtains the best plan of the approaches or one of similar quality (by "similar" we mean within 0.1% of the best solution).

4.3 An Admissible LP-based Heuristic for Goal Utility Dependencies

While we have made efforts toward adapting relaxed plan heuristics for planning problems with goal utility dependencies, there is still a mismatch in terms of optimization. The overall best performing heuristic we have seen so far is inadmissible. Instead, we would like an approach that has more of an optimization perspective. A standard way of setting up a relaxation with an optimization perspective involves (i) setting up an integer programming (IP) encoding for the problem and (ii) computing a linear programming (LP) relaxation of this encoding. In addition to being sensitive to the objectives of the optimization, such a relaxation is also sensitive to more constraints within the problem. In the case of planning, negative interactions between the actions, which is notoriously missing in the standard relaxed plan heuristics, can be accounted for, potentially leading to better heuristic values. One challenge in adopting this approach involves deciding on the exact type of IP encoding for the PSP problem. Although we have experimented with IP encodings for PSP in the previous section, such encodings are better suited for problems with bounded horizons. The normal idea in bounded horizon planning is to put a bound on the number of plan steps. While this idea works for finding feasible plans, it

does not work for finding optimal plans since it is not clear what bound is required to guarantee optimality. We adopt an encoding that is not dependent on the horizon bound. In particular, we describe a compact causal encoding for action selection that accounts for the delete effects of the actions but ignores action ordering. This provides an admissible heuristic.

Our formulation is based on domain transition graphs, first used in the planner Fast Downward [59]. Each of the graphs represents a variable in the multi-valued SAS+ formalism [3] with a value of a variable existing as a vector and effects as arcs between them. We define a network flow problem over each of them. Side constraints are introduced to handle pre-, post-, and prevail-conditions of actions. Additionally, we incorporate parameters, variables, and constraints to handle aspects of goal utility dependencies. Unlike a bounded-horizon (or step) encoding, our encoding is more compact and needs no estimates on plan size for its generation.

After solving for the LP formulation, we can perform a lookahead, similar to what we usually do in our best-first search algorithm when we perform satisficing search (i.e., search using inadmissible heuristics). One difference is that we can extract the relaxed plan using the LP solution as guidance. That is, during a relaxed plan extraction process, if an action is in the LP solution as well as in the planning graph, we select it. This can occasionally improve quality of solutions over a similar lookahead using an relaxed plan extraction process that is directed by cost.

LP Heuristic

We present a novel admissible heuristic that solves a relaxation of the original PSP^{UD} problem by using the LP-relaxation of an IP formulation. We build on the heuristic discussed in [93] for classical planning. While most heuristics ignore the

delete effects of the actions, this heuristic accounts for the delete effects, but ignores action orderings instead. The formulation that we describe is based on the SAS+ planning formalism [3], where a SAS+ planning task is a tuple $\Pi = \langle V, A, s_0, s_* \rangle$ such that $V = \{v_1, \ldots, v_n\}$ represents a set of state variables, A is a finite set of actions, s_0 indicates the initial state and s_* denotes the goal variable assignments. Each $v \in V$ has a domain D_v and takes a single value f from it in each state s, stated as $s[v] = f$. Each action $a \in A$ includes a set of preconditions, $pre(a)$, post-conditions, $post(a)$, and prevail conditions, $prev(a)$.

Previous work has shown that we can translate classical (STRIPS) planning problems into SAS+ planning problems [35, 60], and we use this translation process for generating our heuristic.

We define a SAS+ planning task as a tuple $P = (V, s_0, \mathcal{G}, \mathcal{A})$, where $V = \{v_1, ..., v_n\}$ is a finite set of variables. Each variable $v \in V$ has an associated finite domain D_c. We write $s(v)$ to denote the value of variable v in state s, where s is called a partial state if $s(v)$ is defined for some subset of V, and s is called a state if $s(v)$ is defined for all $v \in V$. s_0 is a state called the initial state and $\mathcal{G}$ is a partial state called the goal. $\mathcal{A}$ is a finite set of actions. Each action $a \in \mathcal{A}$ is of the form $\langle$*pre*, *post*, *prev*$\rangle$, where *pre* and *post* describe the effects of the action and *prev* describes the prevail conditions of the action. We write $eff(a, v)$ to denote the effect of action a in variable v and $prev(a, v)$ to denote the the prevail condition of a in v.

We write $c(a)$ to denote the cost of executing action a, and $u(G_k)$ to denote the utility of achieving goal utility dependency k. The utility of a (partial) state s is given by the sum of all goal utility dependencies satisfied by s. That is, $u(s) =$

$\sum_{k \in K: G_k \in s} u(G_k)$. Our objective is to find a plan π that maximizes *net benefit*, which is given by utility minus cost.

We map this problem into an IP formulation in which the ordering of the actions is ignored. Hence, the formulation is not dependent on the length of the plan and, as a result, only a single IP variable is required for each action. It ignores the ordering of actions and thus is a relaxed formulation of the original problem. After having the IP formulation, which gives an admissible heuristic, we call h_{IP}^{GAI}, we use the solution to its LP relaxation as a further relaxed admissible heuristic that we call h_{LP}^{GAI}. A discussion of the admissibility of the heuristic is found in Appendix A.

The IP formulation models each variable in the planning problem as an appropriately defined network flow problem. Interactions between the variables, which are the result of the action effects and prevail conditions, are modeled as side constraints on the network flow problems. Informally, the formulation seeks to maximize net benefit subject to five sets of constraints: goal constraints, network flow constraints, linking constraints, prevail constraints, and goal utility dependency constraints.

The goal constraints ensure that the hard goals are satisfied, the network flow constraints model the multi-valued fluents, the linking constraints link the action variables with the network flows, the prevail constraints state the conditions for satisfying prevail conditions, and the goal utility dependency constraints state the conditions for satisfying the goal utility dependencies.

Parameters. In order to describe our formulation, we introduce three parameters:

- $cost(a)$: the cost of action $a \in A$.
- $utility(v, f)$: the utility of achieving the value f in state variable v in the goal state.

- $utility(k)$: the utility of achieving the goal utility dependency G_k in the goal state.

Variables. We define five types of variables: (1) Action variables are used to indicate the number of times an action is executed; (2) End value variables are used to indicate which value is satisfied at the end of the solution plan; (3) Effect variables indicate the number of times an effect is executed; (4) prevail variables indicate the number of times a prevail condition is required; and finally, (5) goal dependency variables indicate which goal dependencies are satisfied at the end of the solution plan.

- $action(a) \in \mathbb{Z}^+$: the number of times action $a \in A$ is executed.
- $effect(a, v, e) \in \mathbb{Z}^+$: the number of times that effect e in state variable v is caused by action a.
- $prevail(a, v, f) \in \mathbb{Z}^+$: the number of times that the prevail condition f in state variable v is required by action a.
- $endvalue(v, f) \in \{0, 1\}$: is equal to 1 if value f in state variable v is achieved at the end of the solution plan, 0 otherwise.
- $goaldep(k) \in \{0, 1\}$: is equal to 1 if goal utility dependency G_k is satisfied, 0 otherwise.

Constraints. The constraints are defined as follows:

- Goal constraints for each $v \in V$, $f \in D_v$ such that $f \in G_v$. If f is a goal of v then f must be the end value of v.

$$endvalaue(v, f) = 1 \tag{4.8}$$

- Network flow constraints for each $v \in V$, $f \in D_v$. If a value is deleted n times then it must be added n times. For each variable value there must be a balance of flow (i.e., the number of deletions equals the number additions). If $f \in s_0[v]$ is the initial state of v, then f is added by means of a constant. Similarly, if $f \in G_v$ is a goal, or the end value of v then f is deleted by means of the $endvalue(v, f)$ variable.

$$1\{\text{if } f \in s_0[v]\} + \sum_{\text{effects transition to } f} \textit{effect}(a, v, e) = \sum_{\text{effects that transition from } f} \textit{effect}(a, v, e) + endvalaue(v, f) \tag{4.9}$$

- Linking constraints for each $a \in A$ and $v \in V$. Action variables are linked to their respective effect and prevail variables. Generally there is only one effect or prevail variable per action per variable. Hence, linking constraints would normally be defined as $action(a) = effect(a, v, e)$ or $action(a) = prevail(a, v, f)$. If an action is executed n times, then its effect or prevail condition must be executed n times. The SAS+ formalism, however, allows the precondition of an action to be undefined [3]. We model this by using a separate effect or prevail variable for each possible pre-condition.

$$action(a) = \sum_{\text{effects of } a \text{ in } v} \textit{effect}(a, v, e) + \sum_{\text{prevails of } a \text{ in } v} prevail(a, v, f) \tag{4.10}$$

- Prevail implication constraints for each $a \in A$, $v \in V$, $f \in D_v$. If a prevail condition is executed then the corresponding value must be added at least once. In other words, if there is a prevail condition value f, then f must be

added. We set M to an arbitrarily large value.

$$1\{\text{if } f \in s_0[v]\} + \sum_{\text{effects that transition to } f} \textit{effect}(a, v, e) \geq \tag{4.11}$$

$$\sum_{\text{actions with prevail on } f} prevail(a, v, f)/M \tag{4.12}$$

- Goal dependency constraints for each goal utility dependency k. All values of the goal utility dependency are achieved at the end of the solution plan if and only if the goal utility dependency is satisfied.

$$goaldep(k) \geq \sum_{f \text{ in dependency } k} endvalue(v, f) - (|G_k| - 1) \tag{4.13}$$

$$goaldep(k) \leq endvalue(v, f) \quad \forall f \text{ in dependency } k \tag{4.14}$$

Example: To illustrate the heuristic, let us consider a transportation problem where we must deliver a person, *per1* to a location, *loc2* using a plane, *p1*, and must end with the plan at *loc3*. The cost of flying from *loc1* to *loc2* is 150, from *loc1* to *loc3* is 100, from *loc3* to *loc2* is 200, and from *loc2* to *loc3* is 100. To keep the example simple, we start *per1* in *p1*. There is a cost of 1 for dropping *per1* off. Having *per1* and *p1* at their respective destinations each give us a utility of 1000 (for a total of 2000). Figure 4.3 shows an illustration of the example with each edge labelled with the cost of travelling in the indicated direction (not shown are the utility values for each individual goal).

The optimal plan for this problem is apparent. With a total cost of 251, we can fly from *loc1* to *loc2*, drop off *per1*, then fly to *loc3*. Recall that the LP heuristic, while it relaxes action ordering, works over SAS+ multi-valued fluents. The translation to SAS+ captures the fact that the plane, *p1*, can be assigned to only a single location. This is in contrast to planning graph based heuristics that ignore delete

lists. Such heuristics consider the possibility that objects can exist in more than one location at a given step in the relaxed problem. Therefore, at the initial state, a planning graph based heuristic would return a relaxed plan (RP) that allowed the plane *p1* to fly from *loc1* to *loc2*, and *loc1* to *loc3*, putting it in multiple places at once.

In contrast, the solution from the LP-based heuristic for this problem at the initial state includes every action in the optimal plan. In fact, "1.0" is the value returned for these actions.[3] Though this is a small example, the behavior is indicative of the fact that the LP, through the encoding of multi-valued fluents, is aware that a plane cannot be wholly in more than one place at a time. In this case, the value returned (the *net benefit*, or $2000 - 251 = 1749$) gives us the perfect heuristic.

To use this solution as a candidate in the branch and bound search described in the next section, we would like to be able to simulate the execution of the relaxed plan. For the example problem, this would allow us to reach the goal optimally. But because our encoding provides no action ordering, we cannot expect to properly execute actions given to us by the LP. For this example, it appears that a greedy approach might work. That is, we could iterate through the available actions and execute them as they become applicable. Indeed, we eventually follow a greedy procedure. However, blindly going through the unordered actions leads us to situations where we may "skip" operations necessary to reach the goals. Additionally, the LP may return values other than "1.0" for actions. Therefore, we have two issues to handle when considering the simulation of action execution to bring us to a better state. Namely, we must deal with cases where the LP returns non-integer

[3]The equivalent to what is given by h^{GAI}_{IP}.

values on the action variables and simultaneously consider how to order the actions given to us.

Using an LP for Guidance to Extract a Relaxed Plan: We should only extract plans for sets of goals that appear to be beneficial (i.e., provide a high *net benefit*). We can use the LP for this, as it returns a choice of goals. Given that the LP can produce real number values on each variable (in this case a goal variable), we give a threshold, θ_G on their value. For every goal g, there is a value assignment given by the LP, $Value(g)$. If $Value(g) \geq \theta_G$ then we select that goal to be used in the plan extraction process.

The main idea for extracting a relaxed plan using the LP solution as guidance is to prefer those actions that are selected in the LP solution. When extracting a relaxed plan, we first look at actions supporting propositions that are of the least propagated cost and part of the LP solution. If no such actions support these propositions, we default to the procedure of taking the action with the least propagated cost. Again, since the LP encoding can produce fractional values, we place a threshold on action selection, θ_A. If an action variable $action(a)$, is greater than the threshold, $action(a) \geq \theta_A$, then that action is preferred in the relaxed plan extraction process given the described procedure.

To see why the LP makes an impact on the relaxed plans we extract, let us revisit our ongoing example. Figure 4.4 shows the relaxed planning graph with each action and proposition labeled with the minimum cost for reaching it (using a summing cost propagation procedure). Recall that we want to bias our relaxed plan extraction process toward the actions in the LP because it contains information that the planning graph lacks–namely, negative interactions.

Assume that the LP solver returns the action set {*fly(loc1, loc2)*, *fly(loc2, loc3)*, *drop(p1, loc2)*}. Given that both goals are chosen by the LP, we place both goals into the set of open conditions. We have three layers in the graph, and so we progress backward from layer 3 to 1. We begin with the least expensive goal at the last level and find its cheapest action, *fly(loc1,loc3)*. Since this action is not part of the LP solution (i.e., its value is 0), we move on to the next least expensive supporting action, *fly(loc2,loc3)*. This action is in LP's returned list of actions and therefore it is chosen to satisfy the goal *at(p1,loc3)*. Next, we support the open condition *at(per1,loc2)* with *drop(per1,loc2)*. This action is in the LP. We add the new open condition *at(p1,loc2)* then satisfy it with the action *fly(loc1,loc2)*. We now have the final relaxed plan by reversing the order in which the actions were added. Note that without the LP bias we would have the plan {*fly(loc1,loc2)*, *fly(loc1,loc3)*, *drop(per1,loc2)*}, which is only partially executable in the original planning problem.

Evaluation

We created a planner called BBOP-LP (Branch and Bound Over-subscription Planning using Linear Programming, pronounced "bee-bop-a-loop") on top of the framework used for the planner SPUDS. h_{LP}^{GAI} was implemented using the commercial solver CPLEX 10. All experiments were run on a 3.2 GHz Pentium D with 1 GB of RAM allocated to the planners.

The system was compared against SPUDS and two of its heuristics, h_{relax}^{GAI} and h_{max}^{GAI}. Recall that the heuristic h_{relax}^{GAI} greedily extracts a relaxed plan from its planning graph then uses an IP encoding of the relaxed plan to remove goals that look unpromising. Using this heuristic, it also simulates the execution of the final relaxed plan as a macro action at each state. The other heuristic in SPUDS that we look at,

h^{GAI}_{max}, is admissible and performs max cost propagation (i.e., it takes the maximum reachability cost among supporters of any predicate or action) on the planning graph but does not extract a relaxed plan (and so performs no macro lookahead). It uses the propagated costs of the goals on a planning graph and tries to minimize the set using an IP encoding for the goal utility dependencies.

We use the BBOP-LP system with three separate options. Specifically, we use the h^{GAI}_{LP} heuristic without extracting a relaxed plan for simulation, the h^{GAI}_{LP} heuristic with the LP-based heuristic extraction process, and the h^{GAI}_{LP} heuristic with a cost-based heuristic extraction process. The search terminates only when a global optimal solution is found (or time runs out). A goal and action threshold for the LP-based extraction of 0.01 was used.[4] SPUDS, using an anytime best-first search with the admissible h^{GAI}_{max} heuristic, will also terminate when finding an optimal solution (or a timeout). Note that it is possible that SPUDS using the inadmissible h^{GAI}_{relax} heuristic will terminate without having found an optimal solution (i.e., whenever it chooses to expand a node where $h = 0$). Recall that SPUDS using h^{GAI}_{relax} will also simulate the execution of the relaxed plan. Each of the planners is run with a time limit of 10 minutes.

Problems: We tested our heuristics using variants of three domains from the 3^{rd} International Planning Competition [74]: *zenotravel*, *satellite*, and *rovers*. We use a different reward structure from the problems in our previous tests. The *satellite* and *rovers* have more positive goal utility dependencies, increased reward for individual goals and decreased negative goal utility dependencies. Therefore, these domains are likely to have more positive *net benefit* goal sets than in our previous tests. In

[4] In our experiments, this threshold provided overall better results over other, higher values for θ_A and θ_G that were tested.

zenotravel, moving between locations has a cost about half that of each individual goal reward. We also added more negative goal utility dependencies to this domain.

We tested on the *TPP* domain, but all varieties we attempted returned similarly-valued plans for nearly all of the problems on each of the methods (with a few minor exceptions). Therefore, we do not discuss results for this domain.

Analysis: Figure 4.5 shows the results of running the planners in terms of the *net benefit* of the solutions found and the time it took to search for the given solution value. In 13 of the problems the h^{GAI}_{LP} heuristic with the LP-based relaxed plan lookahead technique performed best. In fact, in only four of the problem instances is this method returning *net benefit* value less than one of the other methods (*zenotravel* problems 14 through 17).

Searching with the h^{GAI}_{LP} heuristic allowed us to find the optimal plan in 15 of the 60 problems, where it exhausted the search space. We contrast this to h^{GAI}_{max}, which exhausted the search space in only 2 of the problems (the first two *zenotravel* problems). However, to the credit of h^{GAI}_{max}, it was able to come close to finding near-optimal solutions in some cases in all of the domains. The new reward structure effectively makes the "best" goal set take longer to reach than in our previous experiments (i.e., it sometimes requires more actions to reach the better goal set). Hence, h^{GAI}_{max} finds plans that give reward in *rovers* unlike in our previous tests, and is unable to find the plans equivalent to h^{GAI}_{relax}. Between h^{GAI}_{max} and h^{GAI}_{LP} (without a lookahead), it turns out that h^{GAI}_{max} gets plans of better net benefit in 3 of the problems in *zenotravel*, 1 problem in *satellite* and 8 problems in *rovers*. However, given the heuristics and search methodology this entails simply collecting more rewards during the search process. Therefore, it's difficult to say how this relates to scalability. However, one advantage h^{GAI}_{LP} has is that it is informed as to

the negative interactions between actions (unlike h^{GAI}_{max} and h^{GAI}_{relax}), so is likely to have a higher degree of informedness (especially as it nears individual goals).

We note that the LP-based relaxed plan lookahead is often better than the other methods (in 13 cases). The differences, however, are usually not significant from the cost-based relaxed plan lookahead. One obvious reason is that both are designed to reach the same LP-selected goals, while the LP-based extracted relaxed plan is informed as to the negative interactions that exist within the problem (e.g., a plane cannot be in more than one place at a time). This has the side-effect that unjustified actions [41] (i.e., actions that do not contribute to the goal) are not considered as often for the lookahead. In our example we saw a best-case scenario of this.

Related, h^{GAI}_{relax} can be fairly accurate in its assessment of which goals to choose, but this can be to its detriment (especially with its way of pruning relaxed plans and performing a lookahead). While it is perhaps ultimately pursuing the "best" subset of goals, if the search cannot actually reach that *complete* subset within the computational time limit, we will not get all reward for it and will likely miss the "second best" goal subset as well. Consider the problem of booking a vacation. A person would want a plane ticket, a hotel reservation, and perhaps a rental car. It is easy enough to see that booking a rental car without the plane ticket or hotel reservation is a foolhardy plan. Stopping short of the entire goal set by getting only the car would be unbeneficial. It turns out that h^{GAI}_{relax}, even with a lookahead, can end up collecting goals that produce negative interactions (through goal utility dependencies and cost dependencies), but over time may be unable to achieve additional goals that can offset this. h^{GAI}_{LP}, while greedier, pursues a larger number of the goals initially. With limited computational time, this can be a better strategy in these problems to find higher quality satisficing solutions. Note that, even in the oc-

casions where h^{GAI}_{LP} is calculated significantly more slowly than h^{GAI}_{relax}, as happens in the more difficult problems of *zenotravel*[5], h^{GAI}_{LP} appears to give better quality plans. This is likely due to its heuristic guidance and/or the lookahead.

4.4 Improving Net Benefit Through Learning Techniques

Use of learning techniques to improve the performance of automated planners was a flourishing enterprise in the late eighties and early nineties, but has however dropped off the radar in the recent years [100]. One apparent reason for this is the tremendous scale-up of plan synthesis algorithms in the last decade fueled by powerful domain-independent heuristics. While early planners needed learning to solve even toy problems, the orthogonal approach of improved heuristics proved sufficiently powerful to reduce the need for learning as a crutch.

However, this situation changing again, with learning becoming an integral part of planning, as automated planners move from restrictive classical planning problems to focus on increasingly complex classes of problems.[6] Like other planning problems, a dominant approach for PSP problems is forward state space search and one challenge in improving these planners has been in developing effective heuristics that take cost and utility dependencies into account. This section of our work [99] aims to investigate if it is possible to boost the heuristic search with the help of learning techniques. Given the optimizing nature of PSP, we were drawn in particular to STAGE [15], which had shown significant promise for improving search in optimization contexts.

[5]For *zenotravel* problem 20, the initial state took 47 seconds (though due to the way the CPLEX solver works, it likely takes much less time per node).

[6]One sign of this renewed interest is the fact that for the first time, in 2008, the International Planning Competition had a track devoted to planners that employ learning techniques. This track was also held in the 2011 International Planning Competition.

STAGE is an online learning approach that was originally invented to improve the performance of random-restart hill-climbing techniques on optimization problems. Rather than resort to random restarts which may or may not help the base-level search escape local minimum, STAGE aims to learn a policy to intelligently generate restart states that are likely to lead the hill-climbing search towards significantly better local optima. The algorithm proceeds in two iterated stages. In the first stage, the base-level hill-climbing search is run until it reaches a local minimum. This is followed by a learning phase where STAGE trains on the sequence of states that the hill-climbing search passed through in order to learn a function that predicts, for any given state s, the value v of the optima that will be reached from s by hill climbing. This learned function is then used in the second stage (alternative) local search to scout for a state s' (that has the highest promise of reaching a better state). If the learner is effective, s' is expected to be a good restart point for the base-level search. The stages are then repeated starting with s' as the initial point.

The main challenge in adapting the STAGE approach to PSP involves finding appropriate state features to drive the learner. In their original work, Boyan and Moore [15] used *hand-crafted* state features to drive learning. While this may be reasonable for the applications they considered, it is infeasible for us to hand-generate features for every planning domain and problem. Moreover, such manual intervention runs counter to the basic tenets of domain-independent planning. Rather, we would like the features to be generated automatically from the problem and domain specifications. To this end, we developed two techniques for generating features. The first uses "facts" of the states and the actions leading to those states as features. The second, more sophisticated idea uses a Taxonomic syntax to generate higher level features [77]. We are not aware of any other work that used the STAGE

approach in the context of automatically generated features. We implemented both these feature generation techniques and used them to adapt a variant of the STAGE approach to support online learning in solving PSP problems. These differ from methods that refine features, such as those done by Fawcett [39]. We compared the performance of our online learning system to a baseline heuristic search approach for solving these planning problems (c.f. [29]). Our results convincingly demonstrate the promise of our learning approach. Particularly, our on-line learning system outperforms the baseline system including the learning time, which is typically ignored in prior studies in learning and planning.

The contributions of this are thus twofold. First, we demonstrate that the performance of heuristic search planners in PSP domains can be improved with the help of online learning techniques. There has been little prior work on learning techniques to improve plan quality. Second, we show that it is possible to retain the effectiveness of the STAGE approach without resorting to hand-crafted features.

In the following sections, we give details of our automated feature generation techniques. Then we show a comparison of the performance of our online learning approach with the baseline heuristic search planner (using h^{GAI}_{relax} but without lookahead techniques as typically used in variants of *Sapa*$^{\mathcal{PS}}$).

Preliminaries

We first provide a few preliminaries on our representation of the problem for our feature generation and on the STAGE approach in general.

Problem Representation: To employ our automatic feature generation methods, we provide a representation of PSP that breaks down the planning problem into components typically seen in domain and problem definitions. Specifically, we define a PSP problem P^o as a tuple of $(O, P, Y, I, \mathcal{G}, U, C)$, where O is a set of

constants, P is a set of available predicates and Y is a set of available action schema. A fact $p \in P$ is associated with the appropriate set of constants in O. $\mathcal{P}$ is a set of all facts. A state s is a set of facts and I is the initial state. Additionally, we define the set of grounded actions A, where each $a \in A$ is generated from $y \in Y$ applied to appropriate set of constants in O. We define actions as we did previously, where each action $a \in A$ consists of precondition $pre(a)$ which must be met in the current state before applying a, $add(a)$ describes the set of added facts after applying a and $del(a)$ describes the set of deleted facts after applying a. C is a cost function that maps an action a to a real valued cost, $C : a \rightarrow \mathcal{R}$. We define our goals $\mathcal{G}$ and utility functions U as in Section 2.

STAGE: STAGE [15] learns a policy for intelligently predicting restart points for a base-level random-restart hill-climbing strategy. It works by alternating between two search strategies, called O-SEARCH and S-SEARCH. O-SEARCH is the base-level local search which hill-climbs with some natural objective function O for the underlying problem (e.g., number of bins used in the bin-packing problem). The S-SEARCH works to scout for good restart points for the O-SEARCH.

The O-SEARCH is run first until, for example, the hill climbing reaches a local minimum. Let $T = s_0, s_1, \ldots, s_n$ be the trajectory of states visited by the O-SEARCH, and let $o_*(s_i) = best_{j>i} O(s_j)$ be the objective function value of the best state found on this trajectory after s_i. STAGE now tries to learn a function V to predict that any state s' that is similar to the state s_i on the trajectory T, will lead the hill-climbing strategy to an optima of value $o_*(s_i)$.

In the next phase, S-SEARCH is run using V as the objective function, to find a state s that will provide a good vantage point for restarting the O-SEARCH. S-SEARCH normally starts from s_n, the state at the end of the trajectory of the previous

O-SEARCH (although theoretically it can start from any random state, including the initial state).[7]

This sequence of O-SEARCH, learning and S-SEARCH are iterated to provide multiple restarts for the O-SEARCH. As we go through additional iterations, the training data for the regression learner increases monotonically. For example, after the O-SEARCH goes though a second trajectory $T_2 : s_0^2, \ldots, s_n^2$ where the best objective value encountered in the trajectory after state s_j^2 is $o_*^2(s_j)$, in addition to the training data from the first O-SEARCH $s_i \rightarrow o_*(s_i)$, we also have the training data $s_j^2 \rightarrow o_*^2(s_j^2)$. The regression is re-done to find a new V function which is then used for driving S-SEARCH in the next iteration.

Boyan and Moore [15] showed that the STAGE approach is effective across a broad class of optimization problems. The critical indicator of STAGE's success turns out to be availability of good state features that can support effective (regression) learning. In all the problems that Boyan and Moore investigated, they provided hand-crafted state features that are customized to the problem. One of the features used for bin-packing problems, for example, is the variance of bin fullness. As we shall see, an important contribution of our work is to show that it is possible to drive STAGE with automatically generated features.

Adapting STAGE to Partial Satisfaction Planning

Automated Feature Generation: One key challenge in adapting the STAGE approach to domain-independent PSP stems from the difficulty in handling the wide variety of feature space between planning domains. While task-dependent features often appear obvious in many optimization problems, domain-independent prob-

[7] In fact, if we can easily find the global optimum of V, that would be the ideal restart point for the O-SEARCH. This is normally impossible because V might be learned with respect to nonlinear (hand-selected) features of state. The inverse image of V on the state space forms its own complex optimization problem, thus necessitating a second local search.

lem solvers (such as typical planning systems) generally require a different set of features for each domain. Producing such features by hand is impractical and it is undesirable to require users of a planning system to provide such a set. Instead, we use automated methods for feature construction.

In our work, we experimented with two methods for feature generation. One method derives propositional features for each problem from the ground problem facts. The other derives relational features for each domain using a Taxonomic syntax [77]. We describe both below. An important difference between Taxonomic and propositional feature sets is that the former remains the same for each domain, while the latter changes from problem to problem even in the same domain. Thus, the number of propositional features grows with the size of problems while Taxonomic features does not.

Propositional Features: In a propositional feature set, each fact in the state represents a feature. Intuitively, if there is some important fact f that contributes to the achievement of some goal or a goal by itself, then states that include the fact should be valued high. In other words, a binary feature that is true with the fact f, should be weighted higher for the target value function. It is then natural to have all the potential state facts or propositions as a feature set. This intuitive idea has been tested in a probabilistic planning system [17]. In their case, the features were used to learn policies rather than value functions. Given constants O and predicates P in a PSP problem P^o, we can enumerate all the ground facts $\mathcal{P}$. Each ground fact is made into a binary feature, with the value of the feature being *true* when the fact is in the current state. We call the planning and learning system that uses these binary features a "Propositional" system.

Relational Features: Although the propositional feature set in the previous subsection is intuitive and a simple method to implement, it cannot represent more sophisticated properties of the domain, where relations between state facts are important, e.g., conjunction or disjunction of the facts.

Our second approach involves relational (object-oriented) features. For many of the planning domains, it is natural to reason with objects in the domain. In particular, it is reasonable to express the value of a state in terms of objects. For example, in a logistics domain, the distance to the goal can be well represented with "number of packages not delivered". Here, the "packages that are not delivered yet" are a good set of objects that indicates the distance to the goal. If we can provide a means to represent a set of objects with such a property, then the cardinality of the set could be a good feature for the value function to learn.

Taxonomic syntax [77] provides a convenient framework for these expressions. In what follows, we review Taxonomic syntax and we define our feature space with Taxonomic syntax.

Taxonomic Syntax: A relational database R is a collection of ground predicates, where ground predicates are applications of predicates $p \in P$ to the corresponding set of objects $(o \in O)$. Each state in a planning problem is a good example for a relational database. We prepend a special symbol **g** if the predicate is from goal description and **c** if the predicate is both true in the current state and the goal state. **c** predicates are a syntactic convenience to express means-ends analysis [78]. Note that goal information is also part of state information. An example relational database (a state from a Logisticsworld domain) is shown in Figure 4.6. In this example, there are two packages *package1* and *package2*. *package2* is not at the

goal location and *package1* is at the goal location. So there is additional fact, (**c**at package1 location1).

Taxonomic syntax C is defined as follows,

$$C = \textbf{a-thing} | (p \;\; C_1 \;\; \ldots \;\; ? \;\; \ldots \;\; C_{n(p)}) | C \cap C | \neg C$$

It consists of **a-thing**, predicates with one position in the argument are left for the output of the syntax, while other positions are filled with other class expressions, intersections of class expressions and negations of a class expression. $n(p)$ is the arity of the predicate p. We define depth $d(C)$ for enumeration purposes. **a-thing** has depth 0 and class expression with one argument predicate has depth 1.

$$d((p \;\; C_1 \;\; \ldots \;\; ? \;\; \ldots \;\; C_{n(p)})) = \max d(C_i) + 1$$

Taxonomic Syntax Semantics: Taxonomic syntax $C[R]$ against a relational database R describes sets of objects. **a-thing** describes all the objects in R. In the example in Figure 4.6, they are (city1, truck1, package1, package2, location1, location2). $(p \;\; C_1 \;\; \ldots \;\; ? \;\; \ldots \;\; C_{n(p)})$ describes a set of objects O that make the predicate p true in R when O is placed in the ? position while other positions are filled with the objects that belong to the corresponding class expression. For example, consider $C = (\textbf{cat} \;\; ? \;\; \textbf{a-thing})$ and let R be the relational database in Figure 4.6. $C[R]$ is then (package1). Among all the objects, only package1 can fill in the $?$ position and make the (**c**at package1 location1) predicate true. Note that **a-thing** allows any object, including location1. As another example, consider $C' = (\text{at} \;\; ? \;\; \textbf{a-thing})$. $C'[R]$ is then (package1, truck1, package2). It is worthwhile to speculate the meaning of C. It indicates all the objects that fill in the first

argument position of **c**at and make the predicate true in the Logisticsworld, which means all the objects that are already in the goal.

Feature Generation Function for Partial Satisfaction Planning: We enumerate limited depth class expressions from the domain definition. **a-thing** is included in the feature set by default. Recall the planning domain definition, $P^o = (O, P, Y, I, G, U, C)$. Using P, the set of predicates, we can enumerate Taxonomic features. First, for all the predicates, except one argument position, we fill all the other argument positions with **a-thing**. This set constitutes the depth 1 Taxonomic features. For the Logisticsworld, C and C' in the above corresponds to this set of depth 1 features. Depth n features can then be easily enumerated by allowing depth $n-1$ Taxonomic syntax in other argument positions than the output position. For example, $(\text{at} \quad \neg(\textbf{c}\text{at} \quad ? \quad \textbf{a-thing}) \quad ?)$ is a depth 2 feature, which is constructed by using a depth 1 Taxonomic feature at the first argument position. The meaning of this feature is "the location where a package is not yet in the goal location". In our experiments, we used depth 2. We could use deeper Taxonomic features, but this increased the solving time during the enumeration and evaluation process. We call the planning and learning system that uses the class expression feature set a "Taxonomic" system. The value of the Taxonomic features is the cardinality of the Taxonomic expressions, which gives out sets of objects. This makes the features appropriate for value function learning.

In both the "Propositional" and "Taxonomic" feature sets, we also use actions involved as part of the features. Each state in PSP includes a trace of the actions that led the initial state to the current state. For the "Taxonomic" feature set, we union these actions with state facts for the relational database construction. The semantics of this database straightforwardly follow from Taxonomic syntax. For

the "Propositional" feature set, we also enumerate all the potential ground actions and assign a binary value 1 if they appear in the actions that led to the state.

Evaluation

To test our approach, we again used variations of domains from the 3rd International Planning Competition (except for TPP). Our experiments use a "vanilla" version of the search with h^{GAI}_{relax} (i.e., it does not perform a lookahead). We used a 2.8 GHz Xeon processor for our tests. For our training data, we used $n = 1000$ evaluated states and set the timeout for each problem to 30 minutes of CPU time [8]. We implemented our system on top of our search framework and used h^{GAI}_{relax} without a relaxed plan lookahead as a baseline search. Note that the learning time was not significant, as the number of automated features generated was typically less than 10,000. This effectively enables our system to perform on-line learning.

To learn from the feature sets, we used a linear regression fit. That is, given our features, we learn a linear function that will output an estimated reward and use this function to determine the "best" reward *net benefit* state from which to restart. To find this function, we used two different libraries for our different automated feature types. The statistical package R [83] was used for the Taxonomic features, but operated more slowly when learning with the binary propositional features. The Java Weka library worked better on this set, and we therefore used it when handling features of this type. For our evaluation, we address the performance of the Stage-PSP system in each domain on the baseline planner [29], Stage-PSP with the Taxonomic features, and Stage-PSP with the propositional features. Note that Stage-PSP systems *include* learning time.

[8] We have tried alternative training data sets, by changing the "n" parameter variously between 500 to 2000, but the results were more or less the same.

For the case of learning with "Taxonomic" features, we also used a simple wrapper method. We greedily add one feature at a time until there is convergence in the approximation measure. For this purpose, we used the R-square metric, which measures the explanation for the variances. This is a practical algorithm design choice for feature selection, since R cannot handle too many features.

Rovers Domain: Figure 4.9 shows the results for this domain. In the graph, the X-axis is for the problem numbers. There were 20 problems. The Y-axis shows net-benefit obtained by each system. As can be seen in the figure, Taxonomic system significantly outperformed SPUDS (using h^{GAI}_{relax} for most of the problems. The *rovers* domain yielded the best results of the three we tested. Except for on a few problem instances, both feature types, the Taxonomic and propositional outperformed SPUDS(with h^{GAI}_{relax}). The cumulative net benefit across the problems in each domain is available in Figure 4.7. In Figure 4.7, for the *rovers* domain, we can see that both of the learning systems, propositional and Taxonomic, outperform the baseline planner, achieving twice the cumulative net benefit of h^{GAI}_{relax} alone. This shows the benefit of the learning involved. Note that, in our experiments, there was no prior training. That is, in most of the recent machine learning systems for planning, they used prior training data to tune the machine learner, while our systems learn online.

Finally, Figure 4.8 lists some of the selected features by the wrapper method with the Taxonomic system. The first listed feature indicates the number of locations traveled where soil data is to be communicated is located. The second provides the number of "take image" actions with rock-analysis in hand. As can be seen in these expressions, the Taxonomic syntax can express more relationally expressive notions than ground facts. Note also that these features make sense:

Moving to a location where soil data will likely move us to improved net benefit. Additionally, taking a goal image while already having finished analysis moves us toward a goal (and therefore higher net benefit).

Satellite Domain: To perform an operation, a satellite needs to turn to the right direction, calibrate its instruments and finally take a photo or perform a measurement. Figure 4.11 shows the results on satellite domain. The performance of Stage-PSP using either of the feature sets does not dominate as strongly as seen in the *rovers* domain. However, Stage-PSP still outperformed the baseline planner in cumulative net benefit measure on the problems, as can be verified through Figure 4.7.

Figure 4.10 lists the features of Taxonomic system found by the wrapper method. The first feature expresses correctly-pointing facts (note that **c**-predicates were used) and the second one expresses the number of actions that turn to the correctly pointing areas, these features help with finding end-state "pointing" goals.

Zenotravel Domain: Figure 4.13 shows the results of *zenotravel* domain. aThe learners did not fare as well in this domain. As can be seen in Figure 4.13, the learning systems lost to SPUDS on the same number of problems as the number of problems they won. The cumulative net benefit across problems is shown in Figure 4.7. The numbers show a slight edge using the Taxonomic features. The margin is much smaller than the other domains.

Figure 4.12 shows the features found in the Taxonomic system. The first feature listed expresses the number of refuel actions taken (and is thus negatively weighted) and the second expresses the number of zooming actions taken to the goal location.

When the learning system fared well, for example, in the *rovers* domain, we found that the learned value function led the S-SEARCH to a quite deeper state s,

that requires many actions to reach from the initial state but achieves the key goal facts.

Although we provided the action features to take the action cost structure into account, the learned value function is not too sensitive to the actions used. One possible reason for this may be that the Taxonomic syntax uses set semantics rather than bag semantics. That is, when the partial plan corresponding to a search node contains multiple instances of an action matching a feature, the action is counted only once.

Summary Motivated by the success of the STAGE approach in learning to improve search in optimization problems, we adopted it to partial satisfaction planning problems. The critical challenge in the adaptation was the need to provide automated features for the learning phase of STAGE. We experimented with two automated feature generation methods. One of them—the Taxonomic feature set—is especially well suited to planning problems because of its object-oriented nature. Our experiments show that our approach is able to provide improvements.

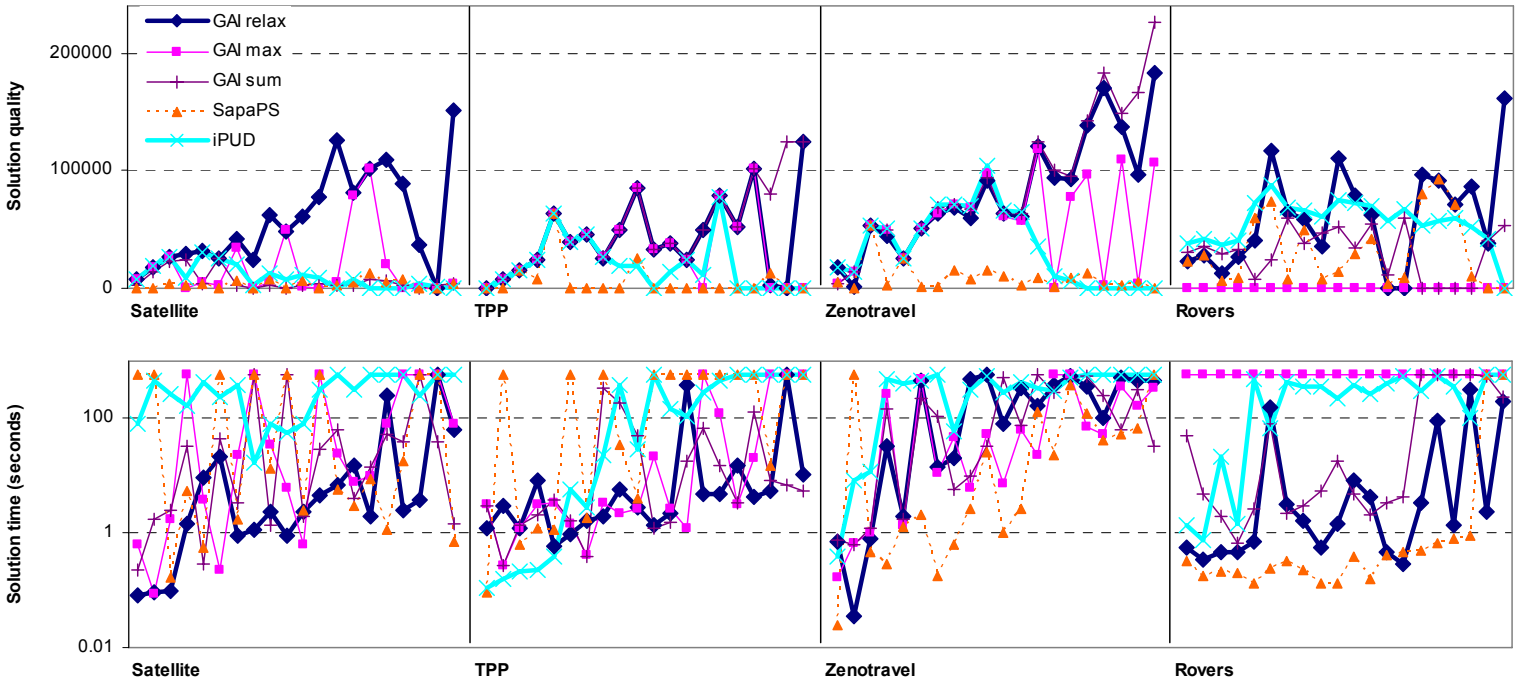


Figure 4.1: Results for goal utility dependency solving methods

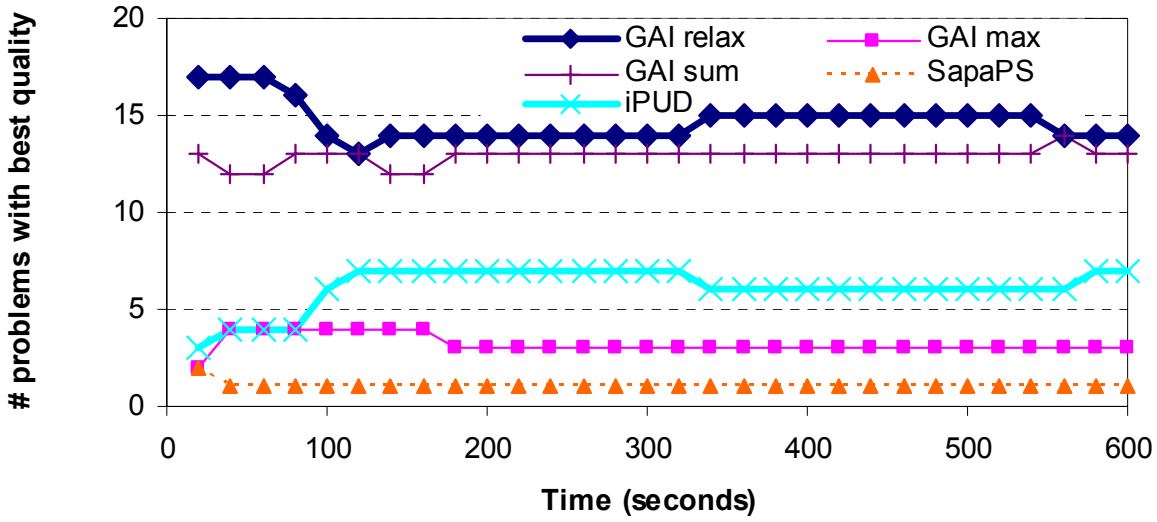


Figure 4.2: The number of highest quality solutions found

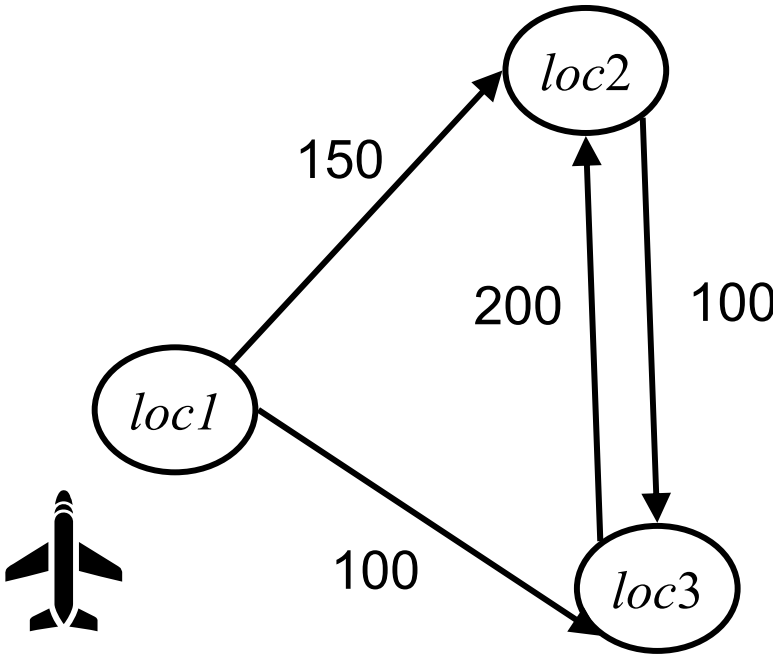


Figure 4.3: A transportation domain example

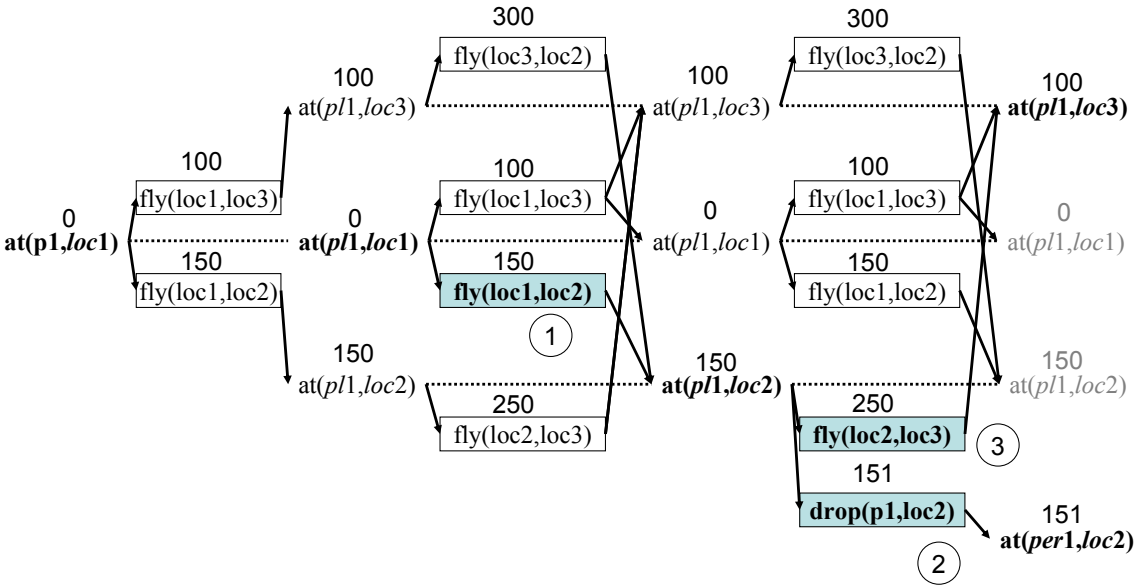


Figure 4.4: A planning graph showing LP-biased relaxed plan extraction

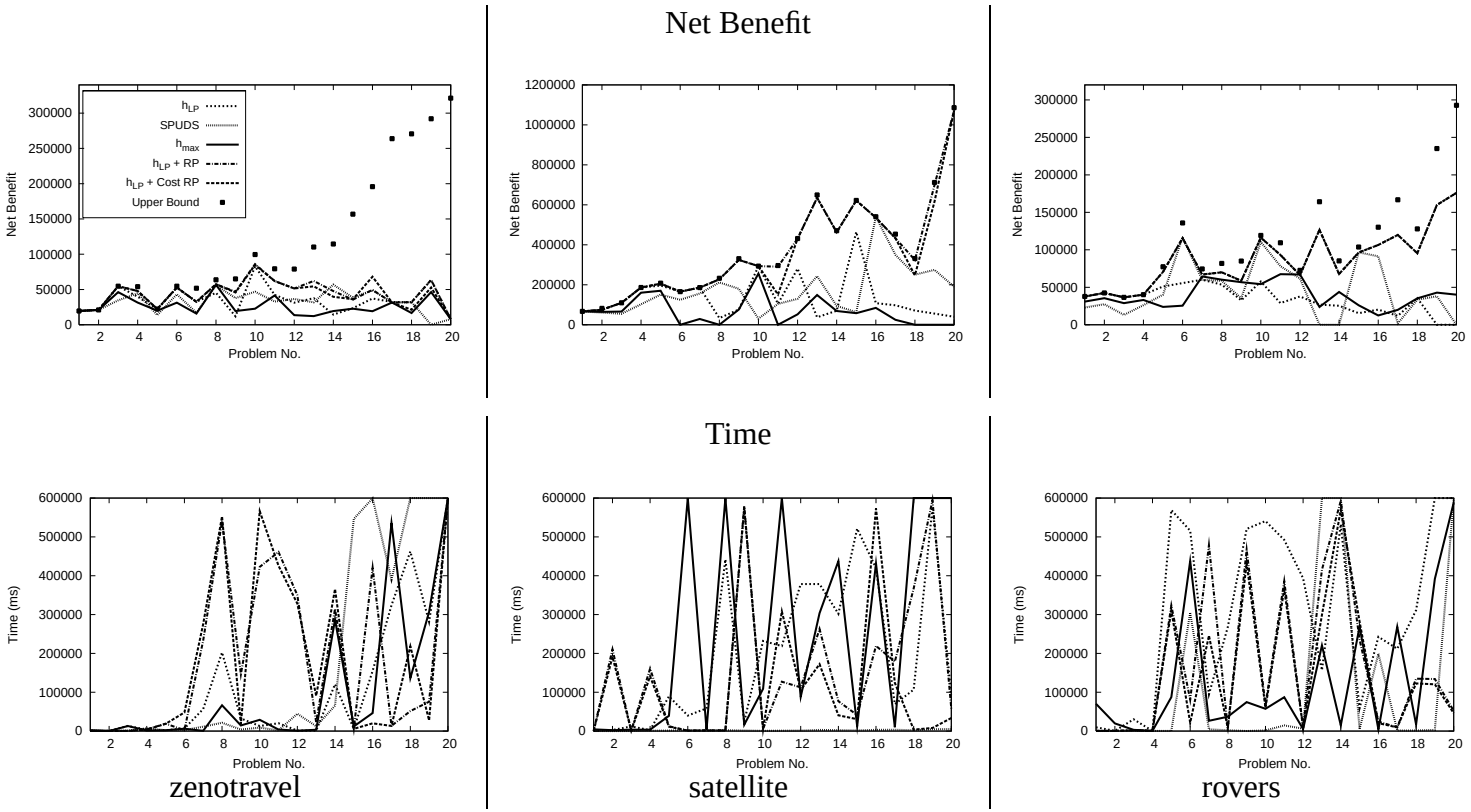


Figure 4.5: Results for the tested domains in terms of total net benefit

(at truck1 location1), (at package1 location1),
(in-city location1 city1), (in-city location2 city1)
(**g**at package1 location1)
(**c**at package1 location1)
(at package2 location2) (**g**at package2 location1)

Figure 4.6: Example Relational Database: A State from Logisticsworld

Domain	Measure	SPUDS	Stage-PSP (Prop)	Stage-PSP (Tax)
Rover	Net Benefit	3.0×10^5	6.0×10^5	6.5×10^5
	No. Features		14336	2874
satellite	Net Benefit	0.89×10^6	0.92×10^6	1.06×10^6
	No. Features		6161	466
zenotravel	Net Benefit	4.3×10^5	4.1×10^5	4.5×10^5
	No. Features		22595	971

Figure 4.7: Summary of the net benefit number of features

(navigate athing (**g**communicated-soil-data ?) ?)

(take-image ? (have-rock-analysis athing ?)
athing athing athing)

Figure 4.8: Taxonomic Features found for Rover domain

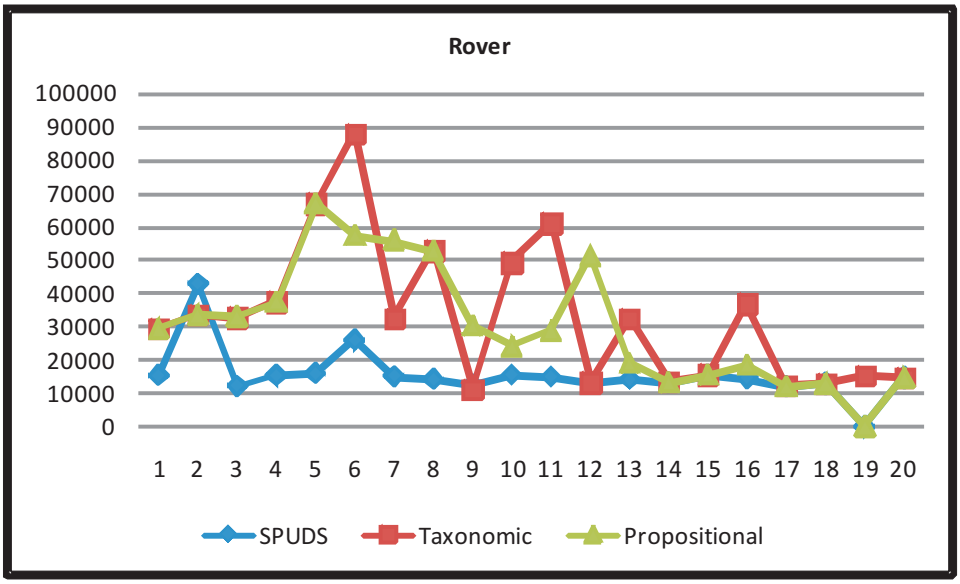


Figure 4.9: Results on rovers domain

(**c**pointing ? athing)

(turn-to (**c**pointing ? athing) athing ?)

Figure 4.10: Taxonomic features found for satellite domain

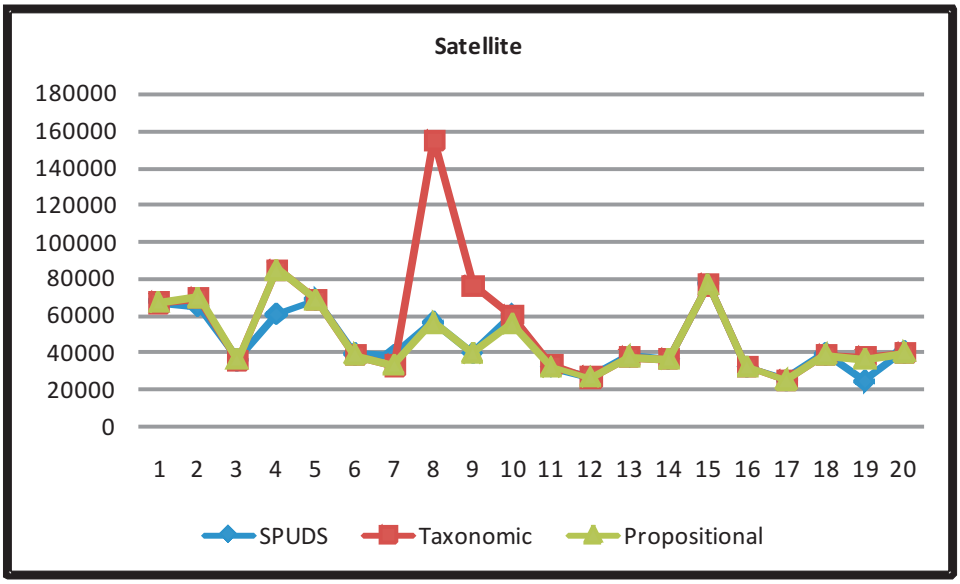


Figure 4.11: Results on satellite domain

(fuel-level ? (fly athing athing athing athing ?))
(**g**at ? (zoom athing athing ? athing athing athing))

Figure 4.12: Taxonomic Features found for zenotravel domain

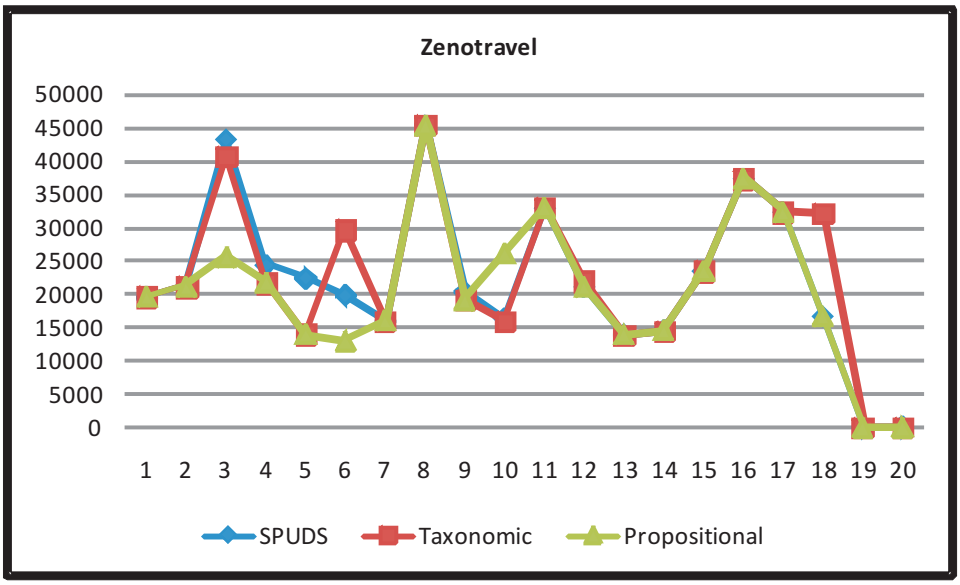


Figure 4.13: Results on zenotravel domain

Chapter 5

PDDL3 "simple preferences" and PSP

While our approach to partial satisfaction planning representations involves assigning rewards for goal achievement, another equivalent approach is to define costs for failing to achieve goals. The organizers of the 5^{th} International Planning Competition (IPC-5) introduced PDDL3.0 [49], which includes this method of defining PSP problems. Indeed, one track named "simple preferences" (PDDL3-SP) has qualities analogous to PSP *net benefit*. Because of the similarity, we studied how our methods could be applied to this representation. Further, we looked whether our planner does better using cost representations alone (i.e., by converting reward to action costs) or if handling rewards directly was a better approach to solving the problem within our framework.

In PDDL3-SP, each preference $p_i \in \phi$ includes a variable $v_{p_i} \in V$ that counts the number of times p_i is violated and $c_i \in \mathcal{C}$ representing the violation cost when p_i is not satisfied. Each action $a \in A$ can have preferences associated with its *precondition*, as can each goal $g \in G$. Additionally, they can include conjunctive and disjunctive formulas on fluents. The objective function is:

$$\text{minimize } c_1 \cdot v_{p_1} + c_2 \cdot v_{p_2} + ... + c_n \cdot v_{p_n} \tag{5.1}$$

where violation costs $c_i \in \mathbb{R}$ are multiplied by the number of times p_i is violated.

We introduce a method of converting PDDL3-SP problems into partial satisfaction planning (PSP) problems, which gives the preferences a reward for achievement rather than a cost for violation. These new problems can then be solved by a

planner capable of solving PSP problems, in our case, we used the planner *Sapa*$^{\mathcal{PS}}$ for a resulting planner we call *Yochan*$^{\mathcal{PS}}$.

There are two main differences between how PDDL3-SP and PSP *net benefit* define *soft* goals. First, in PDDL3-SP, soft goal preferences are associated with a preference name which allows them to be given a violation cost. Second, goal preferences can consist of a disjunctive or conjunctive goal formula. This is opposed to PSP *net benefit* problems where individual goals are given reward. Despite these differences, the similarities are abundant:

- The *violation cost* for failing to achieve an individual goal in PDDL3-SP and *achievement utility* in PSP *net benefit* are semantically equivalent.
- PDDL3-SP and PSP *net benefit* both have a notion of plan quality based on a quantitative metric. PDDL3-SP bases a plan's quality on how well it reduces the goal preference violation cost. On the other hand, PSP *net benefit* views cost as a monotonically increasing value that measures the resources consumed by actions and reward by goal achievement.
- Preferences on action conditions in PDDL3-SP can be viewed as a *conditional cost* in PSP *net benefit*. The cost models on actions differ only in that PDDL3-SP provides a *preference* which acts as a condition for applying action cost.

As part of our compilation, we first transform "simple preference" goals to equivalent goals with utility equal to the cost produced for not satisfying them in the PDDL3-SP problem. Specifically, we can compile a goal preference $pref(G') \mid G' \subseteq G$ to an action that takes G' as a condition. The effect of the action is a newly created goal representing the fact that we "have the preference" $pref(G')$.

The goal compilation process converts goal preferences into additional soft goals and actions achieving them in PSP. We begin by creating a new action a for every preference $pref(G') \mid G' \subseteq G$ in the goals. The action a has G' as a set of preconditions, and a new effect, $g_{G'}$. We then add $g_{G'}$ to the original goal set G, and give it utility equal to the cost $c(pref(G'))$ of violating the preference $pref(G')$. We remove the preference $pref(G')$ from the resulting problem and also force every non-compiled action that destroys G' to remove $g_{G'}$ (by adding $g_{G'}$ to the delete list of these actions).

Other compilation methods for handling the constraints in PDDL3.0 were also introduced in the IPC-5. For instance, the planner MIPS-XXL [36] used a transformation from PDDL3.0 that involved a compilation into hard goals and numeric fluents. *Yochan*$^{\mathcal{PS}}$ and other compilation approaches proved competitive in the competition. In fact, both *Yochan*$^{\mathcal{PS}}$ and MIPS-XXL participated in the "simple preferences" track and received a "distinguished performance" award. However, the compilation used by MIPS-XXL did not allow the planner to directly handle the soft goal preferences present in PDDL3.0. To assist in determining whether considering soft goals directly during the planning process is helpful, we also introduce a separate compilation from PDDL3.0 that completely eliminates soft goals, resulting in a classical planning problem with action costs. The problem is then solved by the anytime A^* search variation implemented in *Sapa*$^{\mathcal{PS}}$. We call the resulting planner *Yochan*$^{\mathcal{COST}}$.

5.1 *Yochan*$^{\mathcal{COST}}$: PDDL3-SP TO HARD GOALS

Recently, approaches to compiling planning problems with *soft* goals to those with *hard* goals have been proposed [36]. In fact, Keyder & Geffner [66] directly handle PSP *net benefit* by compiling the problem into one with hard goals. While

COMPILE-TO-HARD

1. $B := \emptyset$
2. forall $pref(G') \mid G' \subseteq G$
3. create two new actions a_1 and a_2
4. $pre(a_1) := G'$
5. $g_{G'} := name(pref(G'))$
6. $eff(a_1) := g_{G'}$
7. $C(a_1) := 0$
8. $B := B \cup \{a_1\}$
9. $G := (G \cup \{g_{G'}\}) \setminus \{G'\}$
10. $pre(a_2) := \neg G'$
11. $eff(a_2) := g_{G'}$
12. $C(a_2) := c(pref(G'))$
13. $B := B \cup \{a_2\}$
14. $G := (G \cup \{g_{pref}\}) \setminus \{G'\}$
15. $A := B \cup A$

Figure 5.1: PDDL3-SP goal preferences to hard goals.

we explicitly address soft goals in *Yochan*$^{\mathcal{PS}}$, to evaluate the advantage of this approach we explore the possibility of planning for PDDL3-SP by compiling to problems with only hard goals. We call the planner that uses this compilation strategy *Yochan*$^{\mathcal{COST}}$. It uses the anytime A^* search variation from *Sapa*$^{\mathcal{PS}}$ but reverts back to the original relaxed plan heuristic of *Sapa* [31].[1]

Figure 5.1 shows the algorithm for compiling PDDL3-SP goal preferences into a planning problem with hard goals and actions with cost. Precondition preferences are compiled using the same approach as in *Yochan*$^{\mathcal{PS}}$, which is discussed later. The algorithm works by transforming a "simple preference" goal into an equivalent hard goal with dummy actions that give that goal. Specifically, we compile a goal preference $pref(G') \mid G' \subseteq G$ to two actions: action a_1 takes G' as a condition and action a_2 takes $\neg G'$ as a condition (foregoing goal achievement). Action a_1

[1]This is done so we may compare the compilation in our anytime framework.

has cost *zero* and action a_2 has cost equal to the violation cost of not achieving G'. Both a_1 and a_2 have a single dummy effect to achieve a newly created hard goal that indicates we "have handled the preference" $pref(G')$. At least one of these actions, a_1 or a_2, is always included in the final plan, and every other non-preference action deletes the new goal (thereby forcing the planner to again decide whether to re-achieve the hard goal, and again include the necessary achievement actions). After the compilation to hard goals, we will have actions with disjunctive preconditions. We convert these into STRIPS with cost by calling the algorithm in Figure 5.4.

After the compilation, we can solve the problem using any planner capable of handling hard goals and action costs. In our case, we use *Sapa*$^{\mathcal{PS}}$ with the heuristic used in the non-PSP planner *Sapa* to generate *Yochan*$^{\mathcal{COST}}$. We are now *minimizing* cost instead of *maximizing* net benefit (and hence take the negative of the heuristic for search). In this way, we are performing an anytime search algorithm to compare with *Yochan*$^{\mathcal{PS}}$. As in *Yochan*$^{\mathcal{PS}}$, which we will explain in the next section, we assign unit cost to all non-preference actions and increase preference cost by a factor of 100. This serves two related purposes. First, the heuristic computation uses cost propagation such that actions with zero cost will essentially look "free" in terms of computational effort. Second, and similarly, actions that move the search toward goals take some amount of computational effort which is left uncounted when action costs are zero. In other words, the search node evaluation completely neglects tree depth when actions have zero cost.

Example: Consider an example taken from the IPC-5 TPP domain shown in Figure 5.2 and Figure 5.5. On the left side of these two figures we show examples of PDDL3-SP action and goal preferences. On the right side, we show the newly

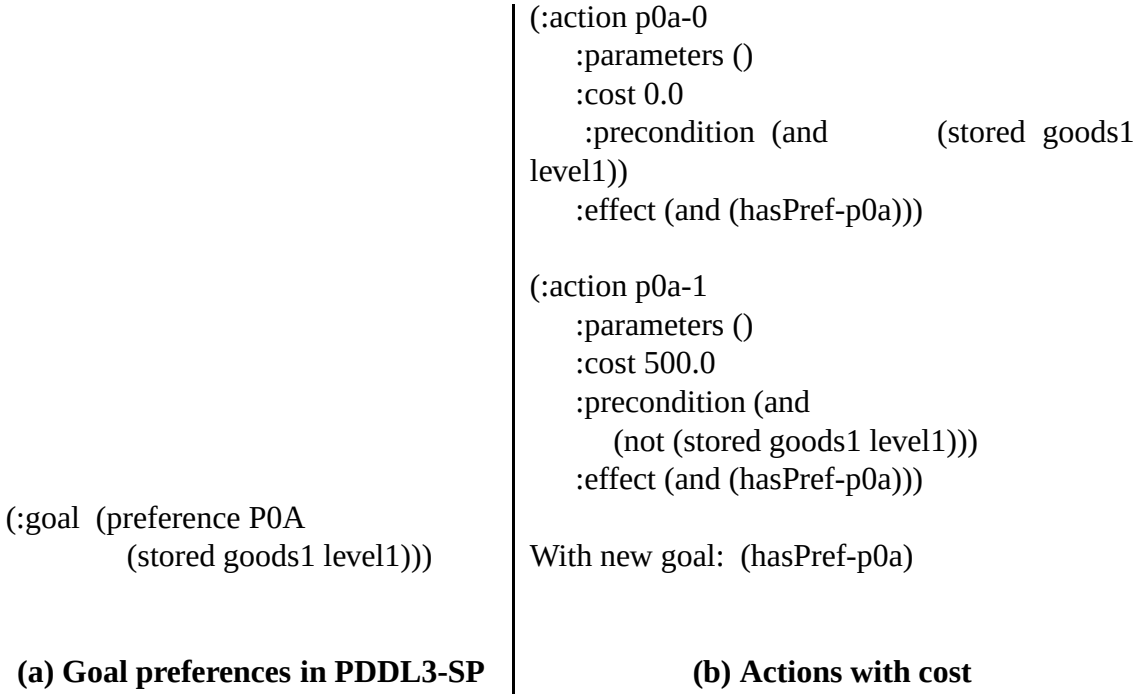


Figure 5.2: PDDL3-SP to cost-based planning.

created actions and goals resulting from the compilation to classical planning (with action costs) using our approach described above.

In this example, the preferred goal `(stored goods1 level1)` has a violation cost of 5 (defined in Figure 5.5). We add a new goal `(hasPref-p0a)` and assign the cost of achieving it with action `p0a-1` (i.e., not having the goal) to 500.

5.2 *Yochan*$^{\mathcal{PS}}$: PDDL3-SP TO PSP

When all soft goals in PDDL3-SP are compiled to hard goals, it is always easiest (in terms of search depth) to do nothing. That is, simply executing the higher cost preference avoidance actions will achieve the goal of having "handled" the preference. Consequentially, the relaxed plan based heuristic may be misleading because it is uninformed of the mutual exclusion between the preference evalua-

tion actions. That is, the heuristic may see what appears to be a "quick" path to a goal, where in fact that path requires the undesirable consequence of violating a preference. Instead, viewing preferences as goals that are desirable to achieve (i.e., attaching reward to achieving them) allows the relaxed plan heuristic to be directed to them. As such, we introduce a method of converting PDDL3-SP problems into PSP problems, which gives the preferences a reward for achievement rather than a cost for violation, thus giving better direction for the relaxed planning graph heuristic. There are two main differences between how PDDL3-SP and PSP *net benefit* define *soft* goals. First, in PDDL3-SP, soft goal preferences are associated with a preference name which allows them to be given a violation cost. Second, goal preferences can consist of a disjunctive or conjunctive goal formula. This is opposed to PSP *net benefit* problems where individual goals are given utility. Despite these differences, the similarities are abundant:

- The *violation cost* for failing to achieve an individual goal in PDDL3-SP and *achievement utility* in PSP *net benefit* are semantically equivalent. Thus, if there is a goal g with a violation cost of $c(g)$ for *not* achieving it in PDDL3-SP, then it is equivalent to having this goal with utility of $u_g = c(g)$ for achieving it in PSP.

- PDDL3-SP and PSP *net benefit* both have a notion of plan quality based on a quantitative metric. PDDL3-SP bases a plan's quality on how well it reduces the goal preference violation cost. On the other hand, PSP *net benefit* views cost as a monotonically increasing value that measures the resources consumed by actions. In PDDL3-SP we have a plan metric ρ and a plan P_1 has a higher quality than a plan P_2 if and only if $\rho(P_1) < \rho(P_2)$. A plan's quality in PSP *net benefit* deals with the trade-off between the utility of the

1. $B := \emptyset$
2. forall $pref(G') \mid G' \subseteq G$
3. $\quad pre(a) := G'$
4. $\quad g_{G'} := name(pref(G'))$
5. $\quad eff(a) := g_{G'}$
6. $\quad B := B \cup \{a\}$
7. $\quad U(g_{G'}) := c(pref(G'))$
8. $\quad G := (G \cup \{g_{G'}\}) \setminus \{G'\}$
9. $\quad$ forall $b \in A$
10. $\quad\quad eff(b) := eff(b) \cup \neg\{g_{G'}\}$
11. $A := B \cup A$

Figure 5.3: Preferences to PSP *net benefit* goals

goals achieved and the cost of the actions to reach the goals. Therefore, a plan P_1 has a higher quality than a plan P_2 in PSP *net benefit* if and only if $U(P_1) - C(P_1) > U(P_2) - C(P_2)$, where $U(P)$ represents the utility of a plan P and $C(P)$ represents the cost of a plan P.

- Preferences on action conditions in PDDL3-SP can be viewed as a *conditional cost* in PSP *net benefit*. The cost models on actions differ only in that PDDL3-SP provides a *preference* which acts as a condition for applying action cost. Like violation costs for goal preferences, action condition violation cost is incurred if a given action is applied to a state where that condition is not satisfied.

As part of our compilation, we first transform "simple preference" goals to equivalent goals with utility equal to the cost produced for not satisfying them in the PDDL3-SP problem. Specifically, we can compile a goal preference $pref(G') \mid G' \subseteq G$ to an action that takes G' as a condition. The effect of the action is a newly created goal representing the fact that we "have the preference" $pref(G')$.

Both PDDL3-SP and PSP *net benefit* have a notion of cost on actions, though their view differs on how to define cost. PSP *net benefit* defines cost directly on each action, while PDDL3-SP uses a less direct approach by defining the penalty for not meeting an execution condition. Therefore, PDDL3-SP can be viewed as considering action cost as a conditional effect on an action where cost is incurred on the preference condition's negation. From this observation, we can compile PDDL3.0 "simple preferences" on actions in a manner that is similar to how conditional effects are compiled [46].

Goal Compilation: The goal compilation process converts goal preferences into additional soft goals and actions achieving them in PSP. Figure 5.3 illustrates the compilation of goals. We begin by creating a new action a for every preference $pref(G') \mid G' \subseteq G$ in the goals. The action a has G' as a set of preconditions, and a new effect, $g_{G'}$. We then add $g_{G'}$ to the original goal set G, and give it utility equal to the cost $c(pref(G'))$ of violating the preference $pref(G')$. We remove the preference $pref(G')$ from the resulting problem and also force every non-compiled action that destroys G' to remove $g_{G'}$ (by adding $g_{G'}$ to the delete list of these actions).

Action Compilation: To convert precondition action preferences, for each action $a \in A$ we generate $P(pref(a))$ as the power set of $pref(a)$ (i.e.,$P(pref(a))$ containing all possible subsets of $pref(a)$). As Figure 5.4 shows, for each combination of preference $s \in P(pref(a))$, we create an action a_s derived from a. The cost of the new action a_s equals the cost of failing to satisfy all preferences in $pref(a) \setminus s$. We remove a from the domain after all of its compiled actions a_s are created. Since some preferences contain disjunctive clauses, we compile them away using the method introduced in by Gazen & Knoblock [46] for converting disjunc-

1. $i := 0$
2. forall $a \in A$
3. foreach $precSet \in P(pref(a))$
4. $pre(a_i) := pre(a) \cup precSet$
5. $eff(a_i) := eff(a)$
6. $c_{a_i} := 100 \times c(pref(a) \setminus precSet)$
7. $A := A \cup \{a_i\}$
8. $i := i + 1$
9. $A := A \setminus \{a\}$

Figure 5.4: Compiling preference preconditions to actions with cost.

tive preconditions in ADL to STRIPS. Notice that because we use the power set of preferences, this could potentially result in a large number of newly formed actions. Since this increase is related to number of preferences, the number of actions that need to be considered during search may seem unwieldy. However, we found that in practice this increase is usually minimal. After completion of the planning process, we apply Equation 5.2 to determine the PDDL3-SP total violation cost evaluation:

$$\text{TOTALCOST} = \sum_{g \in G} u_g - \sum_{g' \in G'} u_{g'} + \sum_{a \in P} c_a \tag{5.2}$$

Action Selection: The compilation algorithm will generate a set of actions A_a from an original action a with $|A_a| = 2^{|pref(a)|}$. Given that actions in A_a appear as separate operators to a planner, this can result in multiple action instances from A_a being included in the plan. Therefore, a planner could produce plans with superfluous actions. One way to fix this issue is to explicitly add negations of the preference conditions that are not included in the new action preconditions (i.e., we can use a negation of the precondition formula in the actions rather than removing the whole condition). This is similar to the approach taken by Gazen & Knoblock [46]

when compiling away conditional effects. This compilation approach, however, may result in several disjunctive preconditions (from negating the original conjunctive preference formula), which will result in even more actions being included in the problem. To overcome this, we use a simple criterion on the plan that removes the need to include the negation of clauses in the disjunctive preferences. Given that all actions in A_a have the same effect, we enforce that for every action generated from a, only the *least cost* applicable action $a_i \in A_a$ can be included in P at a given forward search step. This criterion is already included in *Sapa*$^{\mathcal{PS}}$.

Example: Consider the examples found in Figures 5.5 and 5.6. Figure 5.5 shows the compilation for the TPP domain action: `drive` and Figure 5.6 shows a TPP domain PDDL3-SP goal preference that has been compiled into PSP *net benefit*.

For the action compilation, Figure 5.5 shows the preference `p-drive` has a cost of $10 \times 100 = 1000$ for failing to have all goods ready to load at level 0 of a particular location at the time `drive` is executed. We translate this idea into one where we either (1) have all goods ready to load at level 0 (as in the new action `drive-0` with cost 100) or (2) do not have all goods ready to load at level 1 (as in the new action `drive-1` with cost 1000).

To convert the goal condition from PDDL3-SP into PSP *net benefit* we generate a single action named for the preference, as shown in Figure 5.6. The new action takes the preference goal as a precondition and we again introduce the new goal `(hasPref-p0a)`. However, with this compilation process, we give it a utility value of 5.0. This is the same as the cost for being unable to achieve `(stored goods1 level1)`.

As for implementation details, *Yochan*$^{\mathcal{PS}}$ multiplies the original preference costs by 100 and uses that to direct the forward search. All actions that do not

```
(:action drive
    :parameters
        (?t - truck ?from ?to - place)
    :precondition
        (and
        (at ?t ?from)
            (connected ?from ?to)
            (preference p-drive
                (and
                (ready-to-load
                    goods1 ?from level0)
                (ready-to-load
                    goods2 ?from level0)
                (ready-to-load
                    goods3 ?from level0))))
    :effect (and (not (at ?t ?from))
            (at ?t ?to)))

Weight assigned to preferences:
(:metric
    (+ (× 10 (is-violated p-drive) )
       (× 5 (is-violated P0A) )))
```

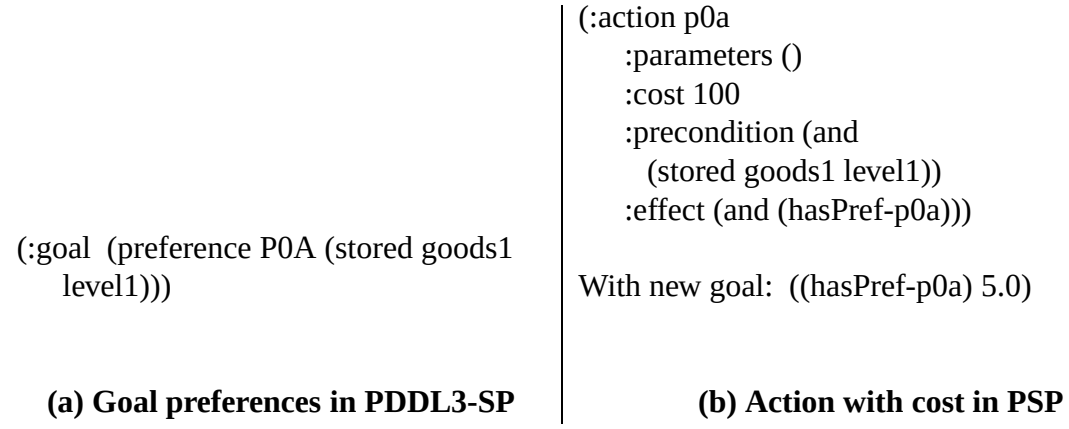
(a) Action preferences in PDDL3-SP

```
(:action drive-0
    :parameters
            (?t - truck ?from ?to - place)
    :cost 100
    :precondition (and
            (at ?t ?from) (connected
                ?from ?to)
            (ready-to-load
                goods1 ?from level0)
            (ready-to-load
                goods2 ?from level0)
            (ready-to-load
                goods3 ?from level0)))
    :effect (and (not (at ?t ?from))
            (at ?t ?to)))

(:action drive-1
    :parameters
            (?t - truck ?from ?to - place)
    :cost 1000
    :precondition (and
            (at ?t ?from) (connected
                ?from ?to))
    :effect (and (not (at ?t ?from))
            (at ?t ?to)))
```

(b) Actions with cost

Figure 5.5: Compiling action preferences from PDDL3-SP to cost-based planning.



(a) Goal preferences in PDDL3-SP **(b) Action with cost in PSP**

Figure 5.6: Compiling goal preferences from PDDL3-SP to PSP.

include a preference are given a default unit cost. Again, we do this so the heuristic can direct search toward short-length plans to reduce planning time. An alternative to this method of artificial scale-up would be to increase the preference cost based on some function derived from the original problem. In our initial experiments, we took the number of actions required in a relaxed plan to reach all the goals at the initial state and used this value to generate a scale-up factor, thinking this may relate well to plan length. However, our preliminary observations using this approach yielded worse results in terms of plan quality.

After the compilation process is done, *Sapa*$^{\mathcal{PS}}$ is called to solve the new PSP *net benefit* problem with the normal objective of maximizing the net benefit. When a plan P is found, newly introduced actions resulting from the compilations of goal and action preferences are removed before returning P to the user.

Evaluation

Most of the problems in the "simple preferences" track of IPC-5 consist of groups of preferred disjunctive goals. These goals involve various aspects of the problems (e.g., a deadline to deliver a package in the *trucks* domain). The *Yochan*$^{\mathcal{PS}}$ compilation

converts each preference p into a series of actions that have the preference condition as a precondition and an effect that indicates that p is satisfied. The utility of a preferred goal is gained if we have obtained the preference at the end of the plan (where the utility is based on the penalty cost of not satisfying the preference in PDDL3-SP). In this way, the planner is more likely to try to achieve preferences that have a higher penalty violation value.

In the competition, *Yochan*$^{\mathcal{PS}}$ was able to solve problems in five of the domains in the "simple preferences" track. Unfortunately, many of the problems in several domains were large and *Yochan*$^{\mathcal{PS}}$ ran out of memory due to its action grounding process. This occurred in the *pathways*, *TPP*, *storage* and *trucks* domains. Also, some aspects of several domains (such as conditional effects and quantification) could not be handled by our planner directly and needed to be compiled to STRIPS. The competition organizers could not compile the *openstacks* domain to STRIPS, and so *Yochan*$^{\mathcal{PS}}$ did not participate in solving it. Additionally, the *pipesworld* domain did not provide a "simple preferences" category. *Yochan*$^{\mathcal{PS}}$ also handles hard goals, which were present in some of the problems, by only outputting plans when such goals are satisfied. The *Sapa*$^{\mathcal{PS}}$ heuristic was also slightly modified such that hard goals could never be removed from a relaxed plan [8].

To test whether varying goal set sizes for the heuristic goal removal process affects our results, we compared running the planner with removing goal set sizes in each iteration of at most 1 and at most 2. It turns out that in almost all of the problems from the competition, there is no change in the quality of the plans found when looking at individual goals (as against individual goals and pairs of goals) during the goal removal process of the heuristic. Only in two problems in the *rovers* domain does there exist a minor difference in plan quality (one in favor of looking

at only single goals, and one in favor of looking at set sizes of one and two). There is also an insignificant difference in the amount of time taken to find plans.

In conclusion, *Yochan*$^{\mathcal{PS}}$ performed competitively in several of the domains given by the organizers of the 5^{th} International Planning Competition (IPC-5). Its performance was particularly good in "logistics" style domains. The quality of the plans found by *Yochan*$^{\mathcal{PS}}$ earned it a "distinguished performance" award in the "simple preferences" track. For comparison, we solved the IPC-5 problems with *Yochan*$^{\mathcal{COST}}$ and showed that compiling directly to classical planning with action cost performs worse than compiling to a PSP *net benefit* problem in the competition domains.

For the rest of this section, we evaluate the performance of *Yochan*$^{\mathcal{PS}}$ in each of the five "simple preferences" domains in which the planner participated. For all problems, we show the results from the competition (which can also be found on the competition website [47]). We focus our discussion on plan quality rather than solving time, as this was emphasized by the IPC-5 organizers. To compare *Yochan*$^{\mathcal{PS}}$ and *Yochan*$^{\mathcal{COST}}$, we re-ran the results (with a small bug fix) using a 3.16 GHz Intel Core 2 Duo with 4 GB of RAM, 1.5 GB of which was allocated to the planners using Java 1.5.

The *Trucks* Domain: The *trucks* domain consists of trucks that move packages to a variety of locations. It is a logistics-type domain with the constraint that certain storage areas of the trucks must be free before loading can take place into other storage areas. In the "simple preferences" version of this domain, packages must be delivered at or before a certain time to avoid incurring a preference violation penalty.

Figure 5.7(a) shows the results for the *trucks* domain in the competition. Overall, *Yochan*$^{\mathcal{PS}}$ performed well in this domain compared to the other planners in the competition. It scaled somewhat better than both MIPS-XXL [36] and MIPS-BDD [36], though the competition winner, SGPlan [64] solved more problems, often with a better or equal quality. Notably, in problems 7 through 9, *Yochan*$^{\mathcal{PS}}$ had difficulty finding good quality plans. Examining the differences between the generated problems provides some insight into this behavior. In the first ten problems of this domain, the number of preferences (i.e., soft goals) increased as part of the increase in problem size. These all included existential quantification to handle deadlines for package delivery, where a package must be delivered before a particular encoded time step in the plan (time increases by one unit when driving or delivering packages). For example, *package1* may need to be delivered sometime before a time step t_3. Because this criterion was defined using a predicate, this caused the number of grounded, soft disjunctive goal sets to increase.[2] This in turn caused more goals to be considered at each time step. The planning graph's cost propagation and goal selection processes would take more time in these circumstances. In contrast, the second set of problems (problems 11 through 20) contained absolute package delivery times on goal preferences (e.g., *package1* must be delivered at exactly time t_5) thereby avoiding the need for disjunctive preferences. The planner solved four instances of these harder problems.[3]

A seeming advantage to *Yochan*$^{\mathcal{COST}}$ in this domain is that it is attempting to find the *least costly* way of achieving the goal set and does not rely on pruning away goals as *Yochan*$^{\mathcal{PS}}$ does. In *trucks*, the violation cost for failing to satisfy goal

[2]Recall that the compilation to PSP *net benefit* generates a new action for each clause of a disjunctive goal formula.

[3]Note that *Yochan*$^{\mathcal{PS}}$ solved more problems than in the competition on the new runs, as the CPU was faster.

preferences turns out to be low for many of the goals, and so the *Sapa*$^{\mathcal{PS}}$ heuristic used by *Yochan*$^{\mathcal{PS}}$ may prune away some of the lower valued goals if the number of actions required for achievement is deemed too high. However, this advantage seems not to help the planner too much here. Also note that *Yochan*$^{\mathcal{COST}}$ has great difficulty with problems 8 and 9. Again, this is largely due to compilation of goals to actions, as the large number of actions that were generated caused the planner's branching factor to increase such that many states with equal heuristic values were generated. When large numbers of preferences exist *Yochan*$^{\mathcal{COST}}$ must "decide" to ignore them by adding the appropriate actions.

The *Pathways* Domain: This domain has its roots in molecular biology. It models chemical reactions via actions and includes other actions that choose initial substrates. Goals in the "simple preferences" track for this domain give a preference on the substances that must be produced by a pathway.

Figure 5.8(a) shows that *Yochan*$^{\mathcal{PS}}$ tends to scale poorly in this domain, though this largely is due to the planner running out of memory during the grounding process. For instance, the number of objects declared in problem 5 caused our grounding procedure to attempt to produce well over 10^6 actions. On most of its solved problems *Yochan*$^{\mathcal{PS}}$ provided equal quality in comparison to the other planners. Figure 5.8(b) shows that both *Yochan*$^{\mathcal{PS}}$ and *Yochan*$^{\mathcal{COST}}$ found plans of equal quality. Note that fixing a small search bug in *Yochan*$^{\mathcal{PS}}$ and *Yochan*$^{\mathcal{COST}}$ caused the planners, in this domain, to fail to find a solution in problem 4 on the new runs (though *Yochan*$^{\mathcal{PS}}$ was able to find a solution during the competition and this is the only problem in which *Yochan*$^{\mathcal{PS}}$ performs worse).

The (IPC-5) *Rovers* Domain: The *rovers* domain initially was introduced at the 3^{rd} International Planning Competition (IPC-3). For the "simple preferences" ver-

sion used in IPC-5, we must minimize the summed cost of actions in the plan while simultaneously minimizing violation costs. Each action has a cost associated with it through a numeric variable specified in the plan metric. The goals from IPC-3 of communicating rock samples, soil samples and image data are made into preferences, each with varying violation cost. Interestingly, this version of the domain mimics the PSP *net benefit* problem in the sense that the cost of moving from place to place causes a numeric variable to increase monotonically. Each problem specifies this variable as part of its problem metric, thereby allowing the variable to act as the cost of traversing between locations. Note that the problems in this domain are not precisely the PSP *net benefit* problem but are semantically equivalent. Additionally, none of the preferences in the competition problems for this domain contain disjunctive clauses, so the number of additional actions generated by the compilation to PSP *net benefit* is small.

As shown in Figure 5.9(a), *Yochan*$^{\mathcal{PS}}$ is able to solve each of the problems with quality that is competitive with the other IPC-5 participants. *Yochan*$^{\mathcal{COST}}$ gives much worse quality plans on three problems and is comparable on the majority of the other problems. For this domain, the heuristic in *Yochan*$^{\mathcal{PS}}$ guides the search well, as it is made to discriminate between goals based on the cost of the actions to reach them. On the other hand, as shown in Figure 5.9(b), *Yochan*$^{\mathcal{COST}}$ attempts to satisfy the goals in the cheapest way possible and, in the harder problems, always returns an empty plan and then fails to find a better one in the allotted time. Thus, *Yochan*$^{\mathcal{COST}}$ tends to find plans that trivially satisfy the newly introduced hard goals.

The *Storage* Domain: Here a planner must discover how to move crates from containers to different depots. Each depot has specific spatial characteristics that must

be taken into account. Several hoists exist to perform the moving, and goals involve preferences for storing compatible crates together in the same depot. Incompatible crates must not be located adjacent to one another. Preferences also exist about where the hoists end up.

In this domain, both *Yochan*$^{\mathcal{PS}}$ and *Yochan*$^{\mathcal{COST}}$ failed in their grounding process beyond problem 5. Figure 5.10(a) shows that, of the problems solved, *Yochan*$^{\mathcal{PS}}$ found solutions with better quality than MIPS-XXL. Figure 5.10(b) shows that both *Yochan*$^{\mathcal{PS}}$ and *Yochan*$^{\mathcal{COST}}$ solved versions of *storage* that had universal and existential quantification compiled away from the goal preferences and produced plans of equal quality. Of the problems solved by both planners, the longest plan found in this domain by the two planners contains 11 actions (the same plan found by both planners).

The *TPP* Domain This is the traveling purchaser problem (TPP), a generalization of the traveling salesman problem. In this domain, several goods exist at various market locations. The object of the planning problem is to purchase some amount of each product while minimizing the cost of travel (i.e., driving a truck) and while also satisfying goal preferences. The *TPP* domain is unique in that it is the only one in the "simple preferences" track to have preference over action preconditions. When driving a truck away from a market, we always prefer to have all of the goods emptied at that market. Cost is added to the action if we fail to satisfy this condition. Like the *trucks* domain, this is a logistics-like domain. Goal preferences typically involve having a certain number of the various goods stored.

As we can see in Figure 5.11(a), *Yochan*$^{\mathcal{PS}}$ finds plans of competitive quality in the problems that were solved. This domain has soft goals that are mutually exclusive from one another (i.e., storing various amounts of goods). Though the

heuristic used in *Yochan*$^{\mathcal{PS}}$ does not identify this, it does focus on finding goals to achieve that may be of the highest quality. It turns out that, in *TPP*, this is enough. As the planner searches for a solution, it identifies this fact and looks for plans that can achieve the highest quality. It is interesting to note that *Yochan*$^{\mathcal{PS}}$ solves more problems than MIPS-XXL and MIPS-BDD. Also, when both find solutions, plans given by *Yochan*$^{\mathcal{PS}}$ are often of better quality.

As Figure 5.11(b) shows, *Yochan*$^{\mathcal{COST}}$ has more difficulty finding solutions for this domain than *Yochan*$^{\mathcal{PS}}$. It attempts to minimize actions as well as cost (as does *Yochan*$^{\mathcal{PS}}$), but tends not to improve plan quality after finding a plan with a lower level of goods (involving fewer actions).

Interestingly, a similarity exists between the anytime behavior of *Yochan*$^{\mathcal{PS}}$ and *Yochan*$^{\mathcal{COST}}$. Typically, both planners discover initial plans at approximately the same rate, and when possible find incrementally better plans. In fact, only when *Yochan*$^{\mathcal{PS}}$ finds better solutions does the behavior significantly differ. And in these cases, *Yochan*$^{\mathcal{PS}}$ "reaches further" for more solutions. We largely attribute this to the heuristic. That is, by ignoring some of the goals in the relaxed plan, the planner essentially serializes the goals to focus on during search. At each search node *Yochan*$^{\mathcal{PS}}$ re-evaluates the reachability of each goal in terms of cost versus benefit. In this way, a goal can look more appealing at greater depths of the search.[4] This is especially noticeable in the *TPP* domain. In this domain, all of the higher-quality plans that *Yochan*$^{\mathcal{PS}}$ found were longer (in terms of number of actions) than those of *Yochan*$^{\mathcal{COST}}$ in terms of number of actions. This is likely because the relaxed plan heuristic in *Yochan*$^{\mathcal{COST}}$ believes preference goals are reachable when they are not.

[4]We also note evidence of this exists by the fact that *Yochan*$^{\mathcal{PS}}$ tends to do better as problems scale-up.

Other Tracks: While *Yochan*$^{\mathcal{PS}}$ participated in the IPC-5 as a partial satisfaction planner capable of handling PDDL3.0, it is based on *Sapa* and therefore is capable of handling a wide variety of problem types. Because of this, the planner also participated in both the "metrictime" and "propositional" tracks. In the "metrictime" track, *Yochan*$^{\mathcal{PS}}$ performed quite well in terms of finding good quality (short makespan) plans, performing best in one domain (the "time" versions of *openstacks*) and second best in three domains (the "time" version of *storage* and *trucks* and the "metrictime" version of *rovers*). The performance in these problems can be attributed to the action re-scheduling procedure of *Sapa*, which takes an original parallel, temporal plan and attempts to re-order its actions to shorten the makespan even more [30]. This especially holds for the *openstacks* problems, whose plans have a high amount of parallelism.

Looking at the results of *Yochan*$^{\mathcal{PS}}$ versus SGPlan for the temporal *openstacks* domain provides some further insight into this behavior. Even in the more difficult problems that *Yochan*$^{\mathcal{PS}}$ solves, the plans contained an equal or greater number of actions. However, *Yochan*$^{\mathcal{PS}}$ parallelized them to make better use of time using its action scheduling mechanism (which, again, was inherited from the planner *Sapa*).

Summary of IPC-5 Results: *Yochan*$^{\mathcal{PS}}$ performs competitively in many domains. In the *trucks* domain, *Yochan*$^{\mathcal{PS}}$ scaled better than MIPS-XXL and MIPS-BDD, but was outperformed overall in terms of number of problems solved by SGPlan, the winner of the competition. There are several technical reasons for *Yochan*$^{\mathcal{PS}}$'s inability to solve large problems in many of the domains: *Yochan*$^{\mathcal{PS}}$'s parsing and grounding routine was quite slow and takes most if not all of the allocated 30 minutes time to parse large problems in many domains.

In three domains (*trucks*, *TPP*, and *rovers*), *Yochan*$^{\mathcal{PS}}$ predominately gave better quality plans than *Yochan*$^{\mathcal{COST}}$. From the search behavior, in many cases the compilation to hard goals caused the planner to quickly choose naïve solutions (i.e., trivially achieving the hard goals without achieving the preference) despite the additional cost associated with doing so. This is attributed to the fact that the heuristic also minimizes the number of actions in the plan while minimizing cost (since the heuristic counts all non-preference actions with a cost 1). While this same quality exists in the heuristic used by *Yochan*$^{\mathcal{PS}}$, handling *soft* goals directly helps the planner by allowing it to completely avoid considering achievement of goals. In other words, the planner can focus on satisfying only those goals that it deems beneficial and can satisfy some subset of them without selecting actions that "grant permission" to waive their achievement.

Note that one issue with *Yochan*$^{\mathcal{COST}}$ is that the number of "dummy" actions that must be generated can affect its search. For every step, the actions to decide to "not achieve the goal" can be applicable, and therefore must be considered (such that a node is generated for each one). This can quickly clog the search space, and therefore results in a disadvantage to the planner as the scale of the problems increases. *Yochan*$^{\mathcal{PS}}$, on the other hand, by directly handling soft goals, can avoid inserting such search states into the space, thereby increasing its scalability over *Yochan*$^{\mathcal{COST}}$.

Interestingly, Keyder and Geffner performed a similar study between cost-based and PSP planners handling compiled versions of problems on domains from the 2008 International Planning Competition [67]. While they did not perform a head-to-head comparison on the same satisficing planner for handling PSP *net benefit* versus handling compiled cost-based versions of the problems, they did show some

benefits. That is, one can use the start-of-the-art in cost-based, satisficing planners through compiling PSP *net benefit* problems into cost-based versions of the problems. Of course, the question of whether we should be handling PSP *net benefit* problems directly or compile them to cost-based planning depends on several factors. For instance, if there are further side constraints related to goal choice that a compilation could not handle, then solving a PSP *net benefit* problem directly would likely be a better choice. Also, planners are likely to respond differently to compiled versions of a problem versus direct handling of goal choice depending upon the techniques they employ.[5]

Up-front Goal Selection in Competition Domains

While *Sapa*^{PS} , and by extension *Yochan*^{PS} , performs goal re-selection during search, one can also imagine dealing with soft goals by selecting them before the planning process begins. Afterward, a planner can treat the selected goals as *hard* and plan for them. The idea is that this two-step approach can reduce the complexities involved with constantly re-evaluating the given goal set, but it requires an adequate technique for the initial goal selection process. Of course, performing optimal goal selection is as difficult as finding an optimal plan to the original PSP *net benefit* problem. However, one can imagine attempting to find a feasible set of goals using heuristics to estimate how "good" a goal set is. But, again, proving the satisfiability of goals requires solving the entire planning problem or at least performing a provably complete analysis of the mutual exclusions between the goals (which is as hard as solving the planning problem).

Given that hard goals must be non-mutex, one may believe that in most domains mutually exclusive soft goals would be rare. However, users can quite easily specify

[5] Since our original comparison, others have also shown other instances where handling PDDL3-SP problems directly can often be better than compilation to cost-based planning [21].

soft goals with complex mutexes lingering among them. For instance, consider a blocks world-like domain in which the soft goals involve blocks stacked variously. If we have three blocks (*a*, *b*, and *c*) with the soft goals *(on a b)*, *(on b c)*, and *(on c a)*, we have a ternary mutual exclusion and we can at best achieve only two of the goals at a time. For any number of blocks, listing every stacking possibility will always generate n-ary mutexes, where n can be as large as the number of blocks in the problem.

Further, the IPC-5 "simple preferences" domains have many n-ary mutual exclusions between goals with sometimes complex interactions such that the satisfaction of one set of goals may be negatively dependent upon the satisfaction of another set of goals (i.e., some goal sets are mutex with other goal sets). It turns out that even when binary mutexes are taken into account, as is done with the planner *AltWlt* (which is an extension of the planner *AltAlt*$^{\mathcal{PS}}$), these complex interactions cannot be detected [85].

Specifically, the planner *AltWlt* uses a relaxed planning graph structure to "penalize" the selection of goals that appear to be binary mutually exclusive by solving for each goal individually, then adding cost to relaxed plans that interfere with already-chosen goals. In other words, given a relaxed plan for a selected goal g called r_g, and a relaxed plan for a candidate goal g', $r_{g'}$, we have a penalty cost c for the selection of g' if any action in $r_{g'}$ interferes with an action in r (i.e., the effects of actions in $r_{g'}$ delete the preconditions found in r_g in actions at the same step). A separate penalty is given if preconditions in the actions of $r_{g'}$ are binary and statically mutex with preconditions in the actions of r_g and the maximum of the two penalties is taken. This is then added to the cost propagated through the

planning graph for the goal. *AltWlt* then greedily selects goals by processing each relaxed plan in turn, and selects the one that looks most beneficial.

To see if this approach is adequate for the competition benchmarks, we converted problems from each of the five domains into a format that can be read by *AltWlt.* We found that in *storage*, *TPP*, *trucks*, and *pathways*, *AltWlt* selects goals but indicates that there exists no solution for the set it selects. However, *AltWlt* found some success in *rovers*, a PSP *net benefit* domain where mutual exclusion between goals is minimal in the benchmark set. The planner was able to solve 16 of the 20 problems, while *Yochan*$^{\mathcal{PS}}$ was able to solve all 20. Of the ones *AltWlt* failed to solve, it explicitly ran out of memory or gave errors. Figure 5.12 shows the results. In 12 of the 16 problems, *AltWlt* is capable of finding better solutions than *Yochan*$^{\mathcal{PS}}$. *AltWlt* also typically does this faster. As an extreme example, to find the eventual final solution to problem 12 of *rovers*, *Yochan*$^{\mathcal{PS}}$ took 172.53 seconds while *AltWlt* took 324 milliseconds.

We believe that the failure of *AltWlt* on the other competition domains is not just a bug, but rather a fundamental inability of its up-front objective selection approach to handle goals with complex mutual exclusion relations. To understand this, consider a slightly simplified version of the simple preferences *storage* domain from the IPC-5. In this domain we have crates, storage areas, depots, load areas, containers and hoists. Depots act to group storage areas into a single category (i.e., there are several storage areas within a single depot). Hoists can deliver a crate to a storage area adjacent to it. Additionally, hoists can move between storage areas within a depot, and through load areas (which connect depots). When a crate or hoist is in a storage area or load area, then no other hoist or crate may enter into

the area. Crates begin by being inside of a container in a load area (hence the load area is initially passable, as no crates are actually inside of it).

Figure 5.13 shows the layout in our example (which is a simplified version of problem 1 from the competition). In the problem there exists a hoist, a crate, a container, two depots ($depot_0$ and $depot_1$) and two storage areas in each depot (sa_{0-0}, sa_{0-1} in $depot_0$ and sa_{1-0}, sa_{1-1} in $depot_1$). The storage areas are connected to each other, and one in each depot is connected to the loading area. The crate begins inside of the container and the hoist begins at in $depot_1$ at sa_{1-0}. We have several preferences: (1) the hoist and crate should end up in different depots (with a violation penalty of 1), (2) the crate should be in $depot_0$ (violation penalty of 3), (3) the hoist should be in sa_{0-0} or sa_{0-1} (violation penalty of 3), (4) sa_{1-0} should be clear (i.e., contains neither the hoist nor the crate with a violation penalty of 2), and (5) sa_{0-1} should be clear (violation penalty of 2).

The (shortest) optimal plan for this problem involves only moving the hoist. Specifically, moving the hoist from its current location, sa_{1-0}, to sa_{0-1} (using 3 moves). This satisfies preference (1) because the crate is not in a depot (hence it will always be in a "different depot" than the hoist), (3) because the hoist is in sa_{0-1}, (4) because sa_{1-0} is clear and (5) because sa_{0-1} is clear. It violates the soft goal (2) with a penalty cost of 3. Of course, finding the optimal plan would be nice, but we would also be satisfied with a feasible plan. However, there is a heavy burden on the goal selection process to find a satisfiable, conjunctive set. In this problem the "simple preference" goals have complex, non-binary mutual exclusions.

Consider the *AltWlt* procedure for finding a set of goals for this domain. *AltWlt* selects goals greedily in a non-deterministic way. But the important aspect of *AltWlt* here is how it defines its penalty costs for noticing mutual exclusion between

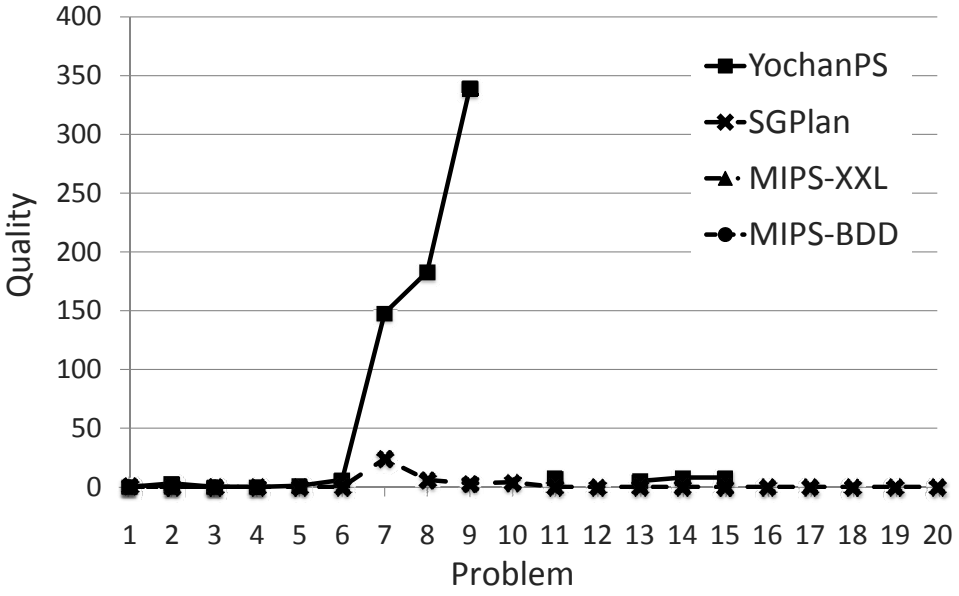
goals. Interference involves the effect of one action deleting the precondition of another action. However, there are often several ways of satisfying a preference, most of which do not interfere with satisfying another preference in the relaxed setting. For instance, consider preference (1), that we should have the create and hoist in different depots. A preference of this form essentially involves several discrete disjunctive clauses, (e.g., "do not have the hoist at sa_{1-1} *or* do not have the crate in $depot_1$"). Satisfying for one of these clauses is sufficient to believe that the preference can be achieved. If we achieve one of these (e.g., "do not have the hoist at sa_{1-1}"), the clause is satisfied. Of course even in the relaxed problem, we must satisfy each of the disjunctive clauses (e.g., we can have each of "do not have the hoist at sa_{x-y} where $x, y \in \{0, 1\}$" or "do not have the crate in $depot_x$ where $x \in \{0, 1\}$"). It turns out that these are satisfiable in the initial state, so this is a trivial feat. If we then choose goal preference (2), having the crate in $depot_0$, we can find a relaxed plan that moves the hoist to the load area, removes the crate from the container and places it in sa_{0-0} (which is in $depot_0$). Satisfying (3), having the hoist at sa_{0-0} or sa_{0-1} looks statically mutex with (1), but the competing needs or interference penalty costs apply only when a relaxed plan exists. Since none exists for (1), *AltWlt* finds a relaxed plan that moves the hoist to sa_{0-1}.[6] Satisfying preference goal (4) requires that we move a single step–easily satisfiable, and sharing an action with (2), and hence there exists no interference or competing needs. Preference goal (5) is satisfied at the initial state.

From this analysis, we can see that *AltWlt* selects each of the goals, as there exist no penalties to make them look unappealing. It will subsequently fail when

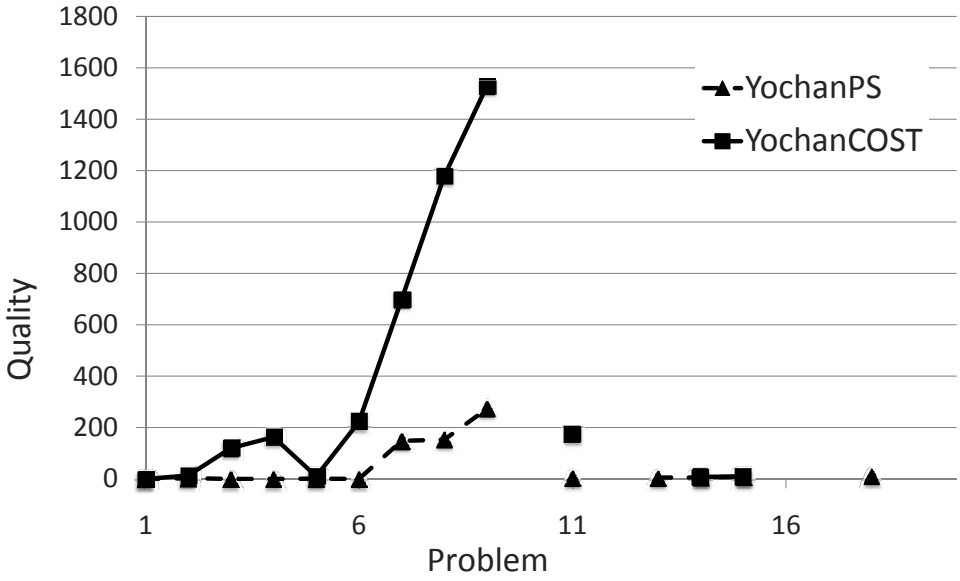
[6]Even if a relaxed plan were to exist for (1), the disjunctive clauses make interference difficult to identify–i.e., we can be satisfying for "do not have the crate in $depot_x$" which is not mutex with preference (3).

attempting to find a solution for the goals—there exists no way to satisfy for all of the preferences. The complex mutual exclusions and disjunctive clauses cause *AltWlt* to select goal sets that are impossible to achieve. From the point of view of the competition, *AltWlt* suffers from similar issues in all but one of the "simple preference" domains (namely, the "simple preferences" version of *rovers*).

In summary, while up-front selection of objectives does allow PSP *net benefit* problems to use other planners, as we have suspected, in complex domains the objective selection cannot even guarantee satisficing plans (beyond the null plan).

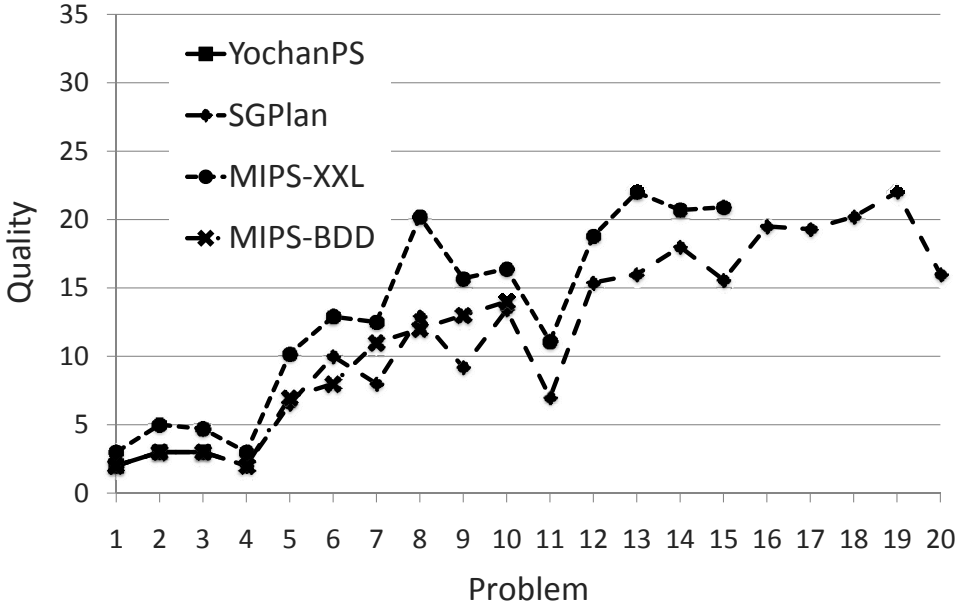


(a) IPC-5 results. *Yochan*$^{\mathcal{PS}}$ solved 13; MIPS-XXL solved 3; MIPS-BDD solved 4; SGPlan solved 20

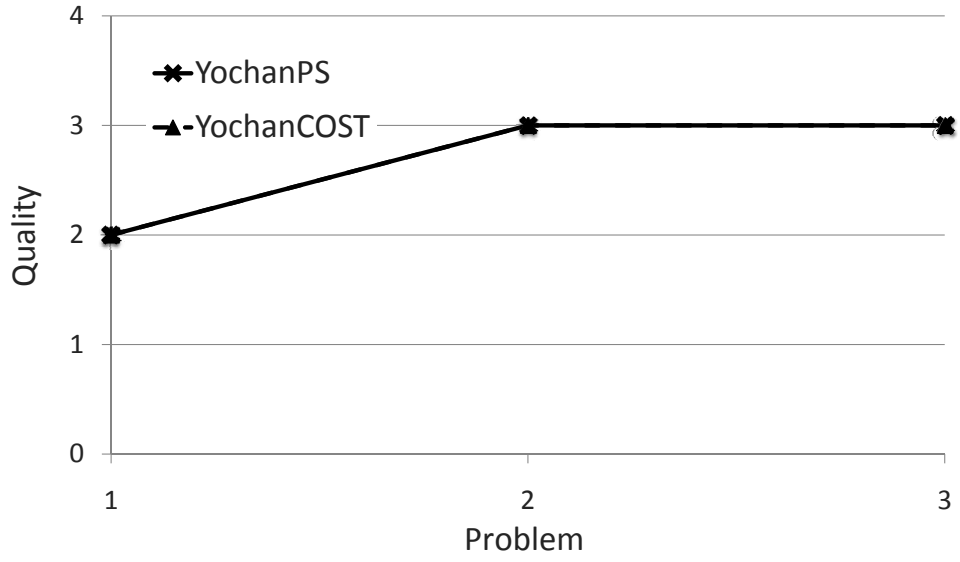


(b) *Yochan*$^{\mathcal{PS}}$ vs. *Yochan*$^{\mathcal{COST}}$. *Yochan*$^{\mathcal{PS}}$ solved 14; *Yochan*$^{\mathcal{COST}}$ solved 12

Figure 5.7: IPC-5 *trucks* "simple preferences"

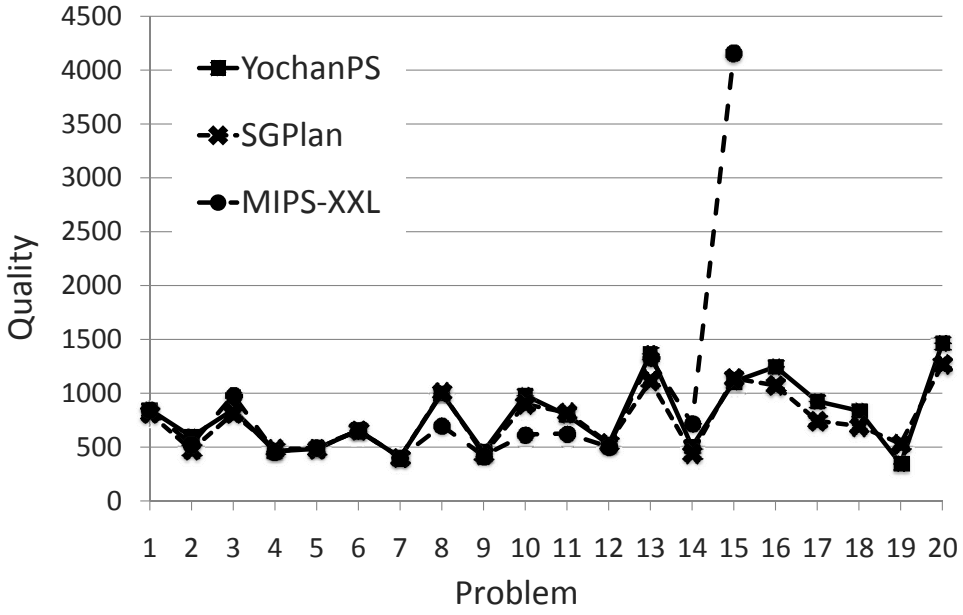


(a) IPC-5 results. *Yochan*$^{\mathcal{PS}}$ solved 4; MIPS-XXL solved 15; MIPS-BDD solved 10; SGPlan solved 30

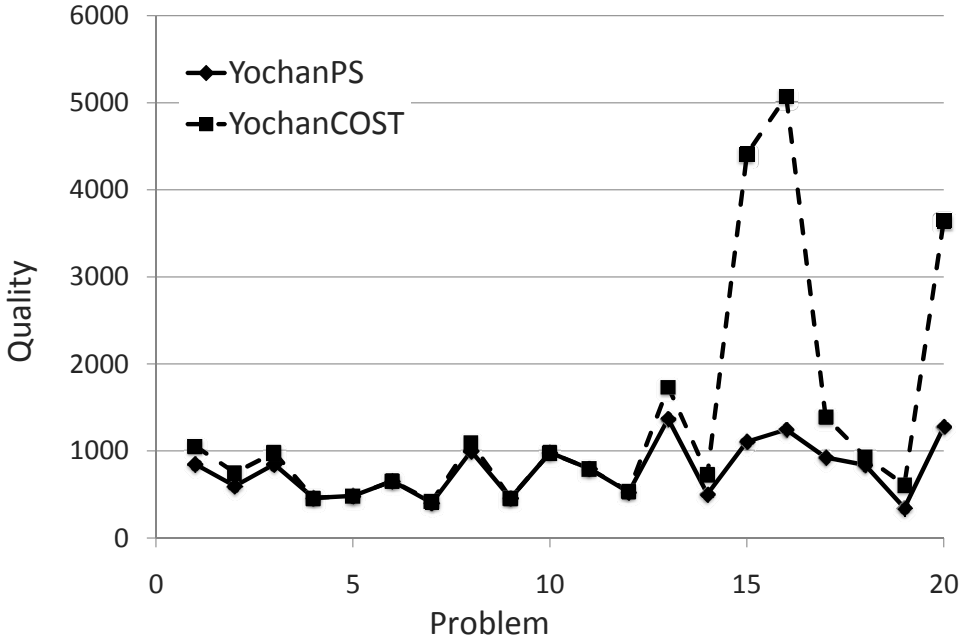


(b) *Yochan*$^{\mathcal{PS}}$ vs. *Yochan*$^{\mathcal{COST}}$. *Yochan*$^{\mathcal{PS}}$ solved 3; *Yochan*$^{\mathcal{COST}}$ solved 3

Figure 5.8: IPC-5 *pathways* "simple preferences"

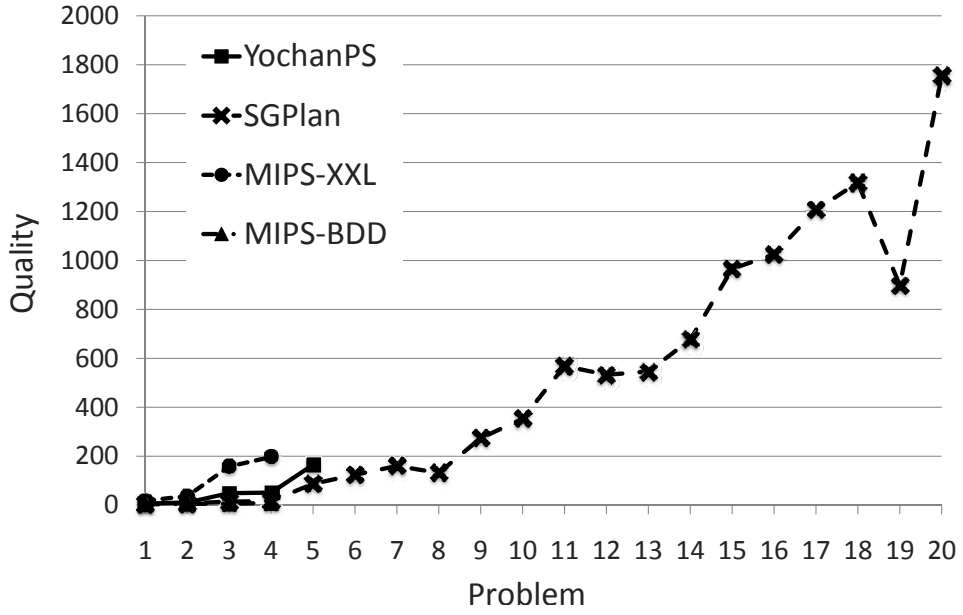


(a) IPC-5 results. *Yochan*$^{\mathcal{PS}}$ solves 20; MIPS-XXL solves 15; SGPlan solves 20

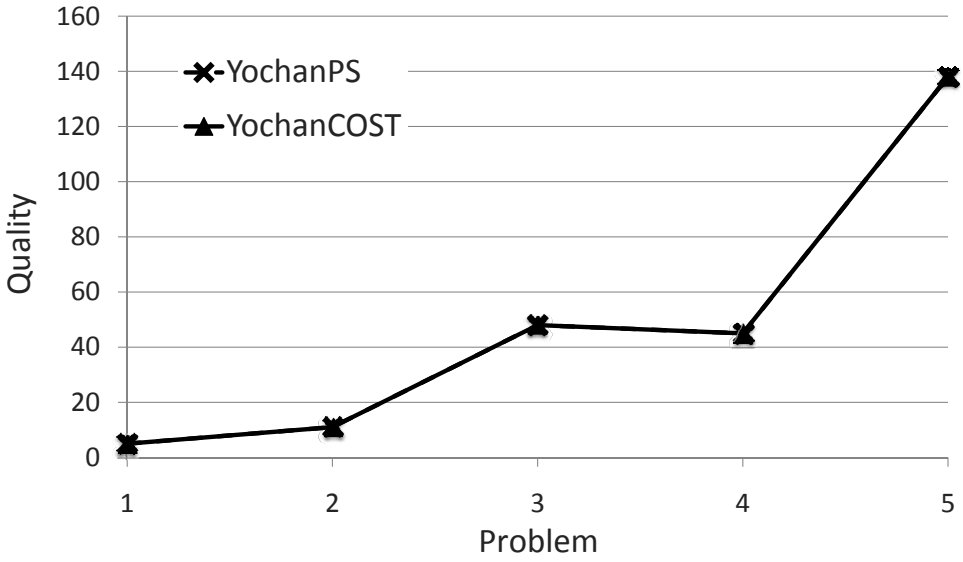


(b) *Yochan*$^{\mathcal{PS}}$ vs. *Yochan*$^{\mathcal{COST}}$. *Yochan*$^{\mathcal{PS}}$ solves 20; *Yochan*$^{\mathcal{COST}}$ solves 20

Figure 5.9: IPC-5 *rovers* "simple preferences"

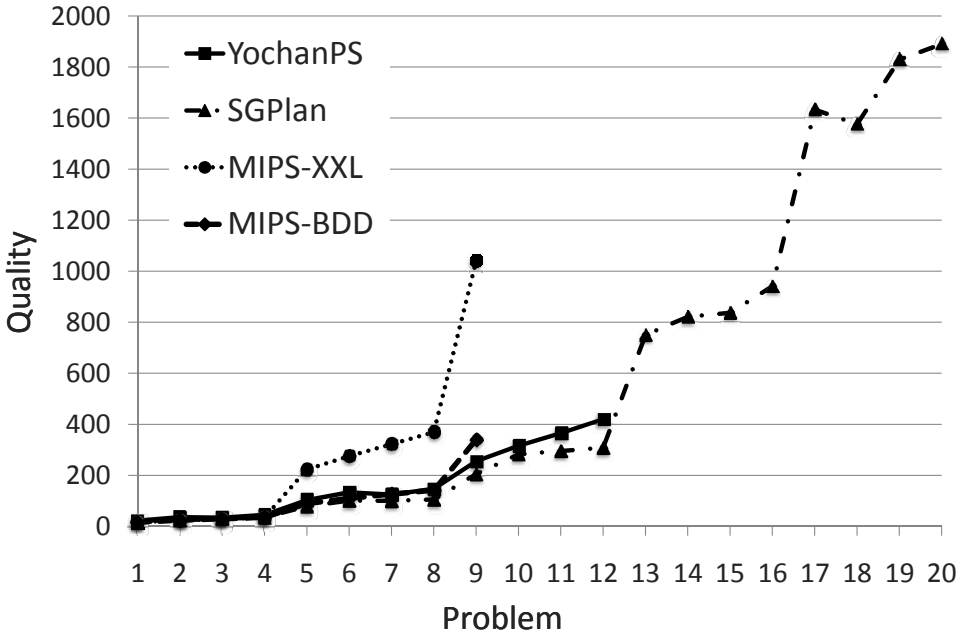


(a) IPC-5 results. *Yochan*$^{\mathcal{PS}}$ solves 5; MIPS-XXL solves 4; MIPS-BDD solves 4; SGPlan solves 20

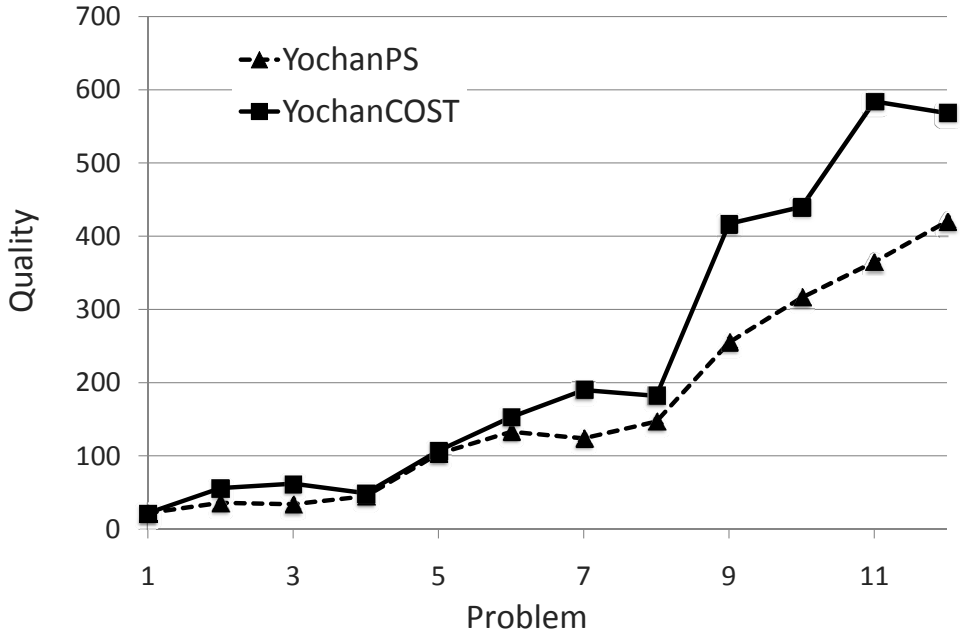


(b) *Yochan*$^{\mathcal{PS}}$ vs. *Yochan*$^{\mathcal{COST}}$. *Yochan*$^{\mathcal{PS}}$ solves 5; *Yochan*$^{\mathcal{COST}}$ solves 5

Figure 5.10: IPC-5 *storage* "simple preferences"



(a) IPC-5 results. *Yochan*$^{\mathcal{PS}}$ solves 12; MIPS-XXL solves 9; MIPS-BDD solves 9; SGPlan solves 20



(b) *Yochan*$^{\mathcal{PS}}$ vs. *Yochan*$^{\mathcal{COST}}$. *Yochan*$^{\mathcal{PS}}$ solves 12; *Yochan*$^{\mathcal{COST}}$ solves 12

Figure 5.11: IPC-5 *TPP* "simple preferences" results

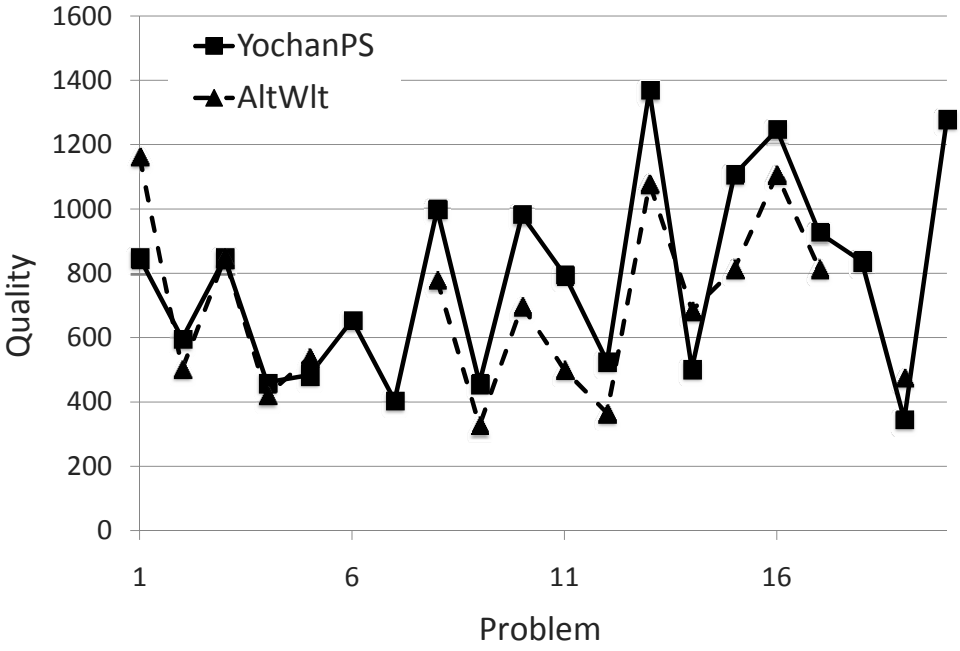


Figure 5.12: Comparison with *AltWlt* on IPC-5 *rovers* domain

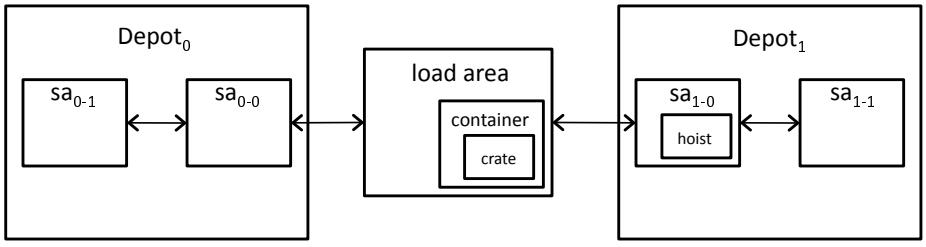


Figure 5.13: An example of the "simple preferences" storage domain

Chapter 6

Time-dependent Goal Achievement Costs

For years, much of the research in temporal planning has worked toward finding plans with the shortest makespan, making the assumption that the utility of a plan corresponds with the time at which it ends. In many problems, however, this does not align well with the true objective. Though it is often critical that goals are achieved in a timely manner, it does not always follow that the shortest plan will be the best in terms of achievement time for *individual* goals. These objectives can occur, for example, when planning for crew activity, elevator operations, consignment delivery, or manufacturing. A few temporal planners (c.f., [52], [23]) are capable of reasoning over similar problems by, for instance, defining hard deadlines. But ranking plans in terms of temporal preferences on plan trajectory or soft deadlines (i.e., those deadlines that can be exceeded, but at a cost) has been less widely explored [36].

The first challenge one faces in considering these problems is how best to represent them so they can be solved. Modeling soft deadlines with a discrete penalty cost, as can be done in the planning domain description language PDDL3, provides an attractive solution to this. In this language, missing a deadline incurs a discrete penalty cost, but discrete models like this have their downsides. With deadlines, for instance, when goal achievement occurs after the deadline point, even by a small amount, the full cost must be paid. This fits some situations—for example, arriving at a ferry terminal after the ferry has left—but it mismatches others, such as being one second late in delivering retail goods. In those cases, once the ideal time for an activity has passed, it is still desirable to achieve the goal at some point,

though preferably sooner. The cost is continuous and *time-dependent*: zero for a certain amount of time, then progressively increasing. In this work, we look toward handling these types of time-dependent, monotonically increasing cost functions.

In dealing with these types of problems, we present techniques that build on POPF [23], a planner particularly well-suited to handling temporal constraints such as soft deadlines due to its rich temporal reasoning engine. This chapter discusses the specifics of how to encode time-dependent cost using a fragment of the planning desciption language PDDL+ [43], and show how a planner can be adapted to support it. In the evaluation we show that the final planner, OPTIC (Optimizing Preferences and TIme-dependent Costs), has state-of-the-art performance on temporal PDDL3 benchmark domains; and that the direct specification of a continuous cost function is not just elegant, but also offers better performance (with search pruning) than if simply compiled to a single sequence of discrete-cost deadlines.

6.1 BACKGROUND: POPF: PARTIAL ORDER PLANNING FORWARD

This work builds on the planner POPF [23], as it offers state-of-the-art temporal planning for planning problems that require concurrency. This is important, because deadlines can induce required concurrency even in problems that could otherwise be solved using action sequences (see [27] for a discussion of these issues). The key distinction between POPF and other forward-chaining temporal planners is that, rather than enforcing a strict total-order on all steps added to the plan, it builds a partial-order plan based on the facts and variables referred to by each step. To support this, each fact p and variable v is annotated with information relating it to the plan steps. Briefly:

- $F^+(p)$ ($F^-(p)$) is the index of the plan step that most recently added (deleted) p;

- $FP^+(p)$ is a set of pairs, each $\langle i, d \rangle$, used to record steps with a precondition p. i denotes the index of a plan step, and $d \in \{0, \epsilon\}$. If d=0, then p can be deleted at or after step i: this corresponds to the end of a PDDL `over all` condition. If d=ϵ, then p can only be deleted ϵ after i.

- $FP^-(p)$, similarly, records negative preconditions on p.

- $V^{\mathit{eff}}(v)$ gives the index of the step in the plan that most recently had an effect upon variable v;

- $VP(v)$ is a set containing the indices of steps in the plan that have referred to the variable v since the last effect on v. A step depends on v if it either has a precondition on v; an effect needing an input value of v; or is the start of an action with a duration depending on v.

The application of actions to states then updates these annotations and, based on their values, produces ordering constraints. Steps adding p are ordered after $F^-(p)$; those deleting p, after $F^+(p)$. Hence, there is a total-ordering on the effects applied to each fact. Preconditions are fixed within this ordering where applying a step with a precondition p orders it after $F^+(p)$; and recording it in $FP^+(p)$ ensures the next deletor of p will ultimately be ordered after it. Similarly, negative preconditions are ordered after some $F^-(p)$ and before the next $F^+(p)$. Finally, steps modifying v are totally ordered, and steps referring to v are fixed within this order (due to effects on v being ordered after the pre-existing $VP(v)$).

An important difference between partially and totally ordered approaches is that the preconditions to support an action are only forced to be true simultaneously if it is added to the plan. Consider a precondition formula F that refers to multiple facts/variables. We can say that $S \widetilde{\models} F$ if the facts/variable values in S support F. If

we apply an action with precondition F we add ordering constraints as discussed above, as otherwise, we could not guarantee the requisite fact/variable values for F are met simultaneously.

For example, consider F=$(a \wedge \neg b)$. In a state where $S\widetilde{\models}F$ it is possible that another action, B^+, adding b can be applied after its last deletor, $F^-(b)$. Since the last adder of a, $F^+(a)$, is not necessarily ordered with respect to either $F^-(b)$ or B^+ the plan may be scheduled such B^+ is before $F^+(a)$, and thus $a \wedge \neg b$ is not necessarily true at any point. The key point here is that visiting a state S_i where $S_i\widetilde{\models}F$ is not sufficient to guarantee F will be satisfied during the plan.

6.2 Planning with Continuous Cost Functions

In considering problems with continuously changing cost on goals, there are two key challenges:

1. How to best represent planning problems where the value of a plan rests with the time individual goals are achieved.

2. Given a representation, how to solve these problems.

In addressing the first point, this work explores how to use PDDL3 to represent discretizations of the continuous cost function, and representing cost functions directly using a combination of PDDL+ and cost evaluation actions. The semantics of PDDL3 offer an all-or-nothing approach to cost, requiring the generation of a set of deadlines for the same goal, giving a piece-wise representation of the original cost function. This may be sufficient (or even accurate) for many problems. For example, the London Underground system operates on a fixed schedule, where making a stop 5 minutes late may be no worse than being 3 minutes late; either way the train

will depart at the same time. But in other problems, it leaves open questions on the granularity of cost deadlines.

As an example, consider a simple logistics problem where blueberries, oranges and apples must be delivered to locations, B, O and A respectively. Each fruit has a different shelf-life. From the time they are harvested, apples last 20 days, oranges 15 days and blueberries 10 days. The truck has a long way to travel, driving with the perishable goods from an origin point P. Let us assume equal profit for the length of time each item is on a shelf. The time to drive between P and B is 6 days, between P and A is 7 days, between B and O is 3 days, and between A and B is 5 days. To make all deliveries, the shortest plan has a duration of 15 days; that is, drive to points A, B, then O in that order. If we were to deliver the goods in this order, the blueberries and oranges will rot before they reach their destinations, and the total time-on-shelf for the apples would be 13 days. Instead, we need a plan gets the best overall value. A plan that drives to point B, O, then A achieves this, though it does so in 17 days. In this case, summing the total time-on-shelf across all fruits gives us 15 days. Given a fruit type f and a shelf-life, sl_f (in days), we can create a set of deadlines such that the cost increases by $1/sl_f$ each day.

An unfortunate disadvantage of this approach is that it may improperly represent costs; for example, missing the deadline by only a few moments would immediately place the cost in the next day "bracket", an overly strict requirement for this problem. In this case, a more direct approach to representing cost is desirable. Therefore, we also consider cost represented by a continuous, monotonically increasing function, comprising arbitrary piecewise monotones expressible in PDDL. In this representation, cost is zero until time point t_d, then increases continuously until it reaches a cost c at a time point $t_{d+\delta}$ (see Section 2.2). This approach re-

moves issues of granularity for the domain modeler when they are not required. However, the question is which model is better in practice. And we shall see later in this chapter that while representing cost functions discretely has disadvantages, it also has benefits in terms of solving time which we can migrate over to solving for continuous representations (generating a hybrid, tiered search approach).

Continuous Cost Functions in PDDL+

We now look at how to model continuous cost functions using PDDL+ [43] without reference to deadlines in PDDL3. First, in order to track the time elapsed throughout the plan, we introduce a variable (`current-time`), assigned the value 0 in the initial state. This is updated continuously by a process with no conditions and the effect (`increase (current-time) (* #t 1)`), increasing the value of current-time by one per time-unit. As processes execute whenever their conditions are met, and in this case the condition is tautologous, one can now write actions whose effects are dependent on the time at which they are executed.

For each goal fact g_i upon which one wants to enforce a time-dependent cost, one adds a fact `goal-`g_i to the initial state, and replaces the goal with a fact `collected-`g_i. Then, it is possible to create an action following the template in Figure 6.1; the action can have arbitrary parameters, as required by the goal, and the cost function can differ for different goals. The line marked with `*` is optional, depending on the semantics required. For goals that should persist after the cost has been collected, the line is present; otherwise, it is not. The conditional effects of the example increases the variable `total-cost` by a linear formula if `current-time` is after `deadline-one-`g_i (i.e., t_d), but before `final-deadline-`g_i and by a fixed amount of `current-time` is after `final-deadline-`g_i (i.e., $t_{d+\delta}$). This corresponds with the definition from Section 2.2. With additional conditional effects (i.e., intermedi-

```
(:action collect-goal-g1 :parameters (?p1 ?p2 - obj)
:precondition (and  (goal-g1 ?p1 ?p2) (g1 ?p1 ?p2))
:effect (and (collected-g1 ?p1 ?p2)
             (not (goal-g1 ?p1 ?p2))
             (not (g1 ?p1 ?p2))
 (when (> (current-time) (final-deadline-g1 ?p1 ?p2))
      (increase (total-cost)  (full-penalty ?p1 ?p2)))
 (when (and (> (current-time) (deadline-one-g1 ?p1 ?p2))
        (<= (current-time) (final-deadline-g1 ?p1 ?p2))))
 (increase (total-cost)
  (* (full-penalty ?p1 ?p2)
   (/ (- (current-time) (deadline-one-g1 ?p1 ?p2))
   (- (final-deadline ?p1 ?p2) (deadline-one-g1 ?p1 ?p2))
)))))
```

Figure 6.1: Structure of a cost-collection action for time-dependent cost

ate deadlines), the cost function can consist of an arbitrary number of stages, each taking the form of any mathematical function expressible in PDDL. If restricting attention to cost functions that monotonically increase (i.e., problems where doing things earlier is always better), any reasonable cost-aware planner using this model will apply such actions sooner rather than later to achieve minimal cost.

Comparison to Discretized Model

The cost functions above (omitting the asterisked effect) have a PDDL3 analog. In theory, it is possible to obtain the same expressive power by creating a sequence of several discrete PDDL3 deadline (i.e., `within`) preferences, with the spacing between them equal to the greatest common divisor (GCD) of action durations, and each with an appropriate fraction of the cost. In other words, we can define a step function approximation of the cost function using the GCD to define cost intervals. This could give a substantial blow-up in the size of many problems. A more coarse discretization with the discrete deadlines spaced further apart than the GCD may be more practical. However, a planner using such a model may also fail to reach optimal solutions; it may be possible to achieve a goal earlier but not sufficiently

early to achieve the earlier deadlines, so the planner will not recognize this as an improved plan.

Solving for Time-dependent Continuous Costs

The new planner, OPTIC, handles these problems by extending the POPF scheduler, heuristic and the search strategy. The planner also makes a small extension to handle the very basic type of PDDL+ process needed to support the `current-time` ticker. Specifically, processes with static preconditions and linear effects on a variable defined in the initial state (but not subsequently changed by the effect of any other actions). Supporting these requires very little reasoning in the planner.

Scheduling: The compilation (in the absence of support for events) requires that all cost functions be monotonically increasing. Given this (and the absence of preferences and continuous numeric change, other than the ticker) a simple temporal problem (STP) [28] scheduler suffices; the lowest cost for a given plan can be achieved by scheduling all actions at their earliest possible time, and so can represent the plan as a simple temporal problem as in the original POPF planner. The earliest time for each action can be found by performing a single-source shortest path (SSSP) algorithm on the temporal constraints of a plan. When a `collect-`g_i action is first added to the plan, the planner increases the recorded plan cost according to its cost function evaluated at its allotted timestamp. Subsequently, if the schedule of a plan moves `collect-`g_i to a later timestamp, the cost of the plan is increased to reflect any consequential increase in the cost function of the action.

Admissible Heuristic: Now that it is possible to compute the cost of solutions, a heuristic can be used to guide search toward finding high-quality solutions; and ideally, an admissible heuristic that can be used for pruning. In satisficing planning, relaxed plan length has been a very effective heuristic [63], and OPTIC uses

this to guide search. The planner continues to use this for its search (as done in the other planners we discuss in this dissertation), but it also uses a second, admissible, heuristic for pruning. Each reachable `collect-cost` action yet to be applied will appear in a temporal relaxed planning graph (TRPG). In OPTIC's TRPG, one can obtain an admissible estimate of each `collect-`g_i's achievement time by using its cost at the action layer in which it appears. Since costs are monotonically worsening, this cost is an admissible estimate of the cost of collecting the associated goal. Since `collect-`g_i actions achieve a goal which is never present as a precondition of an action, and they have numeric effects only on cost, they fit the model of *direct-achievement costs* used in the heuristic of POPF [22]. Thus, the sum of the costs of the outstanding collect actions, at their earliest respective layers, is an admissible estimate of the cost of reaching the remaining goals.

Tiered Search: While searching for a solution, the planner can use the admissible estimate h_a for pruning. In general, it can prune a state s, reached by incurring cost $g(s)$ (as computed by the scheduler), with admissible heuristic cost $h_a(s)$, if $g(s) + h_a(s) \geq c$, where c is an upper-bound on cost (e.g., the cost of the best solution so far). If the granularity of cost is N, then states are kept if $g(s)+h_a(s) \leq c - N$. In the case of PDDL3, where exceeding deadlines incurs a discrete cost, N is the cost of the cheapest preference. When searching with continuous time-dependent costs, though, N is arbitrarily small, so the number of such states is large. Hence, compared to the discrete-cost case, the planner is at greater risk of exhausting the available memory. If one inflated N, then more states could be pruned. However, this forfeits optimality, effectively returning to the discretized case.

As a compromise, it may be better to use a tiered search strategy. Specifically, one can invoke WA* a number of times in sequence, starting with a larger value of N and finishing with $N{=}\epsilon$ (some small number). The principle is similar to IDA* [70], and reminiscent of iterative refinement in IPP [69], but applied to pruning on plan quality. That is, it is possible to introduce an aggressive bound on cost, i.e., assume there exists a considerably better solution than that already found; if this does not appear to be the case, then one can gradually relax the bound. The difference from IDA* comes in the heuristic value used for search. Since the planner still uses relaxed plan length to guide search, we use the admissible cost-based heuristic and cut-off value only for pruning.

6.3 EVALUATION

No benchmarks with continuous cost functions exist so we created some based on existing problems; namely, Elevators, Crew Planning and Openstacks, from IPC-2008. In Elevators, the objective is to bring people to their final floors using different elevators capable of reaching various floors at differing speeds. The deadlines were generated based on greedy, independent solutions for each passenger, thereby generating a "reasonable" wait time for the soft deadline and a partially randomized "priority" time for when full cost is incurred (with the idea that some people are either more important or more impatient than others.) For each of problems 4–14 from the original problem set (solvable by POPF), there were three problems generated. In Crew Planning, the goals involve a crew performing various tasks. In this domain, for each problem solvable by POPF (1–20), we generated soft deadlines on each crew member finishing sleep, and random deadlines for payload tasks each day. In Openstacks, a manufacturing-based domain, each original problem is augmented by soft deadlines based on production durations.

The critical question to answer is whether supporting continuous costs is better than using a discretization comprising a series of incremental deadlines (modeled using PDDL3). Thus, for each continuous model several discretized problems, with each continuous cost function approximated by either 3, 5 or 10 preferences (10 being the closest approximation), were generated. This is compared to OPTIC with the continuous model, and either normal search (only pruning states that cannot improve on the best solution found), or the tiered search described in Section 6.2. In the latter, the value of N was based on the cost Q of the first solution found. The tiers used were $[Q/2, Q/4, Q/8, Q/16, \epsilon]$. Each tier had at most a fifth of the 30 minutes allocated. The results are shown in Figure 6.2, the graphs show scores calculated as in IPC-2008; i.e. the score on a given problem for a given configuration is the cost of the best solution found (by any configuration) on that problem, divided by the cost of its solution.

First, observe that the solid line, denoting tiered search, has consistently good performance. Compare this to continuous-cost search without tiers; it is worse sometimes in Elevators, often in Crew Planning, and most noticeably in Openstacks. These domains, in left-to-right order, have a progressively greater tendency for search to reach states that could potentially be *marginally* better than the incumbent solution; risking exhausting memory before reaching a state that is *much* better. This is consistent with the performance of the most aggressive split configuration, where we split the continuous cost function into three discrete all-or-nothing penalty deadlines. In Elevators, and some Crew Planning problems, its aggressive pruning makes it impossible for it (or the other split configurations) to find the best solutions. But, looking from left-to-right between each graph, the memory-saving benefits of this pruning become increasing important, and by Openstacks, it is find-

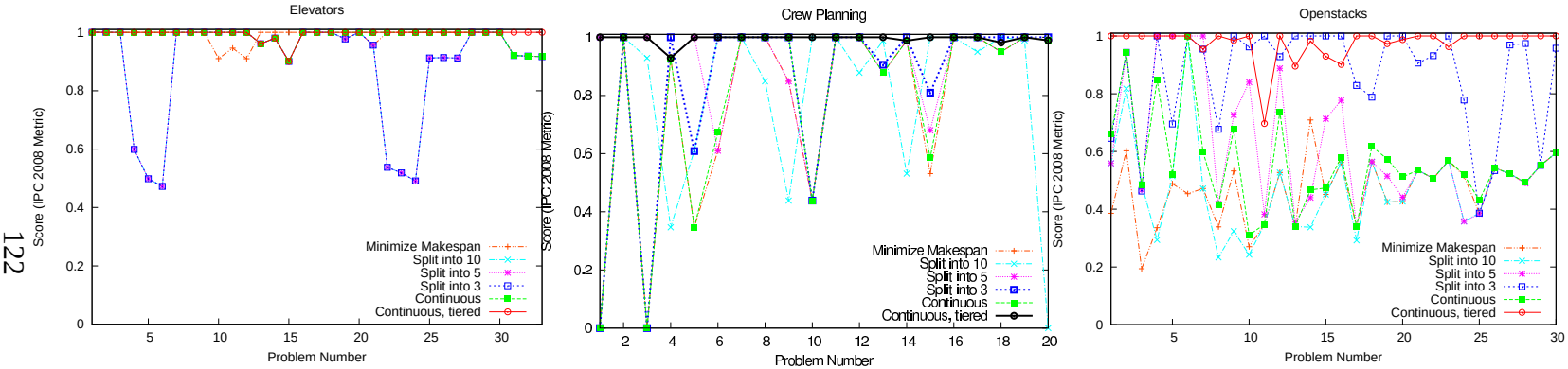
ing better plans. Here, too, the split configurations with weaker pruning (5 and 10) suffer the same fate as non-tiered continuous search, where memory use limits performance.

From these data, it is clear that the benefit of tiered-search is that it is effectively performing dynamic discretization. Because we have modeled continuous-costs in the domain, rather than compiling them away, the "improvement requirement" between successive solutions becomes a search-control decision, rather than an artifact of the approximation used. In earlier tiers, search prunes heavily, and makes big steps in solution quality. In later tiers, pruning is less zealous, allowing smaller steps in solution quality, overcoming the barrier caused by coarse pruning. This is vital to close the gap between a solution that is optimal according to some granularity, but not globally optimal. A fixed granularity due to a compilation fundamentally prevents search from finding the good solutions it can find with a tiered approach.

Finally, note that plan makespan is not always a good analog for plan cost. In Elevators, it appears to be reasonable (likewise in the PDDL3 encoding of the Pipesworld domain earlier in the evaluation). In Crew Planning and Openstacks, though, we see that minimizing makespan produces poor quality solutions; indeed in Openstacks, low makespan solutions are particularly bad.

Summary We have considered temporal planning problems where the cost function is not directly linked to plan makespan and explored how to handle temporal problems with continuous cost functions that more appropriately model certain classes of real-world problems and gone on to show the advantages of reasoning with a continuous model of such problems versus a compilation to PDDL3 via discretization. Our tiered search approach appears to offer the benefits of the discretized

representation while operating over the continuous representation of the planning problem.



(a) Elevators (b) Crew planning (c) Openstacks

Figure 6.2: IPC scores per problem, validated against the continuous cost domain

Chapter 7
Related Work

While there has been recent growth in research for solving planning problems in the presence of soft (prioritized) goals, such problems have been discussed in the field of artificial intelligence before. Indeed, Simon discussed issues of goal choice, relating it to computation time, cognition and motivation [87]. However, only in the last several years has there been a significant effort in using modern planning technology to solve problems of this nature.

Issues like how to best represent soft goals, whether compilation approaches are always effective in all PSP problems, and the level of expressiveness required for solving real-world problems with soft goals are still open questions. Representing PSP problems is the foremost problem, and a variety of approaches have been proposed. For instance, work has been done in defining qualitative goal preferences, soft constraints on plan trajectories and explicit declarations on resource limitations. The solving methods range from various heuristic approaches, compilations to other problem substrates (e.g., integer programming or boolean formula satisfiability) or compilations that reduce soft goal constraints to planning with other objectives. In this chapter, we review these methods.

7.1 Representations for Partial Satisfaction Planning

For atemporal goal satisfaction, much of this dissertation focuses on both goal *cost* and *utility* dependencies. We use the *general additive independence* model for representing goal utility dependencies, but there are several other attractive models. In particular, the *UCP-Network* model [14] allows one to specify preference relationships between goals with an associated reward for their satisfaction. This model

focuses on *conditional* dependencies (i.e., if one already has an airplane ticket to Hawaii, then one will get reward for having a hotel reservation in Hawaii). Another similar approach is to use the *graphical model* [2]. While both of these provide a graphical representation that can make it easier for users to understand (and define) dependencies, GAI is more general and both of these models can be compiled to GAI.

The languages from the 2006 and 2008 International Planning Competitions, PDDL3 [48] and PDDL3.1 (respectively), can also be used to represent goal utility dependencies. Indeed, they are capable of representing more complex preferences over trajectories and therefore focus on a broader problem class than ours. Only one domain from the planning competitions, *woodworking* (from 2008), contains well-defined utility dependencies between goals. Even in this domain, those utility dependencies are strictly positive, making goal choice much easier than if negative utility dependencies existed. Indeed, it is unclear whether any planner capable of handling PDDL3 dependencies can deal with negative dependencies (our preliminary testing shows that they cannot, though it may be possible to easily force them to).

Qualitative preference-based planners also treat goals as soft constraints; however, goals are not quantitatively differentiated by their utility values, but their preferences are instead qualitatively represented. Qualitative preferences are normally easier to elicit from users, but they are less expressive and there can be many plans generated that are incomparable. Brafman and Chernyavsky [16] use TCP-Networks to represent the qualitative preferences between goals. Some examples are: (1) $g_1 \succ g_2$ means achieving g_1 is preferred to achieving g_2; (2) $g_1 \succ \neg g_1$ means achieving g_1 is better than not achieving it. Using the goal preferences, plan

P_1 is considered better than plan P_2 if the goal set achieved by P_1 is preferred to the goal set achieved by P_2 according to the pre-defined preferences. A Pareto optimal plan P is the plan such that the goal set achieved by P is not dominated (i.e., preferred) by the goal set achieved by any other plan. Tran and Pontelli [92] introduced the PP language that can specify qualitative preferences on plan trajectories such as preferences over the states visited by the plan or over actions executed at different states. PP uses a nested subset of temporal logic (similar to PDDL3) to increase the set of possible preferences over a plan trajectory. PP is later extended with quantification and variables by Bienvenu et al. [10].

7.2 Planners Solving PSP and Their Close Relatives

There are several planners that solve PSP and closely related problems, and they fall into three distinct strategies: (1) up-front goal selection; (2) combined goal and action selection (i.e., planning directly on action and goal selection); (3) compilation into another substrate (e.g., cost-based planning, integer programming or boolean formula satisfiability). All of these approaches try to solve the problem of choosing among the $2^{|\mathcal{G}|}$ possible goal sets in different ways.

Up-front Goal Selection: An appealing method is to perform goal selection up-front and find a reasonable plan for those goals then either stop or find another goal set to try to get an even better plan. This is a two-step strategy, where in step one we heuristically select a subset of soft goals and in step two we convert the goal set into hard goals then use a non-PSP solving method to find a solution for those goals. This lets you use an "off-the-shelf" planner for finding solutions. The planners *SGPlan* [64], the orienteering-planner (OP) [88], *AltAlt*$^{\mathcal{PS}}$ [95] and HSP$^*_p$ [57] all use this type of strategy. *SGPlan* performs an up-front goal selection that has not been well-described in the literature, though it iterates through all soft goals and

uses a heuristic to choose a "best" goal set. It then solves the problem using its regular search. In its first step, OP uses the solution of a simpler problem to select both the subset of goals and the order to achieve them. The abstract problem is built by first propagating the action costs on the planning graph and constructing the *orienteering* problem, which is a variation of the *traveling salesman* problem. The approach was used to find a solution with limited resources, and similar approach was used by García-Olaya, et al. [45] in their work on the same problem. Note that the *orienteering* problem has similarities to the flow-network IP formulation we use the planner BBOP-LP for the h^{GAI}_{LP} heuristic.

Unlike the orienteering-planner, *AltAlt*$^{\mathcal{PS}}$ relies on the cost-sensitive planning graph and uses a different technique to analyze the graph to heuristically select the most beneficial subset of goals. After the goals are found, *AltAlt*$^{\mathcal{PS}}$ uses a variation of the regression search planner *AltAlt* to search for a low cost plan. HSP$^*_p$ works somewhat differently. It iterates through all soft goal sets and uses IDA* [70] to solve the goal set it decides is best. On each iteration of IDA*, it chooses a "best" goal set that gives the current highest bound plan quality using its heuristic. This can be seen as a mixed strategy between up-front goal selection and performing goal selection during search.

The disadvantage of this approach is that if the heuristics in the first step do not select the right set of goals then the planner may either find a poor quality plan or can take a lot of time to discover that the problem is unsolvable before it can switch to another goal set. Therefore, if the first step does not select the *exact* optimal goal set, then the final plan is not guaranteed to be optimal. Moreover, if there is an unachievable goal selected, then the planner will return in failure (with some planners trying to select another set of goals after this). Indeed, as shown

in Section 5.2, *AltAlt*$^{\mathcal{PS}}$ and its improved version *AltWlt* never try to solve more than a single (hard) goal set and can consistently select the set of goals containing non-obvious mutexes on many problems.

Combined Action and Goal Selection: Our approaches fall into this category. The SPUDS and BBOP-LP heuristics perform goal and action selection before they return a heuristic value. Several other PSP planners perform this type of search. Of course, the planner we have based our work on, *Sapa*$^{\mathcal{PS}}$, does this [7] (as well as its PDDL3-SP variant, *Yochan*$^{\mathcal{PS}}$). It uses a goal selection technique during search (i.e., per state). The planners MIPS-XXL [36], MIPS-BDD [34], Gamer [37], and HPlan-P [5] also perform goal selection during the planning process. With the exception of Gamer, these planners use a forward search. MIPS-XXL [36] and MIPS-BDD [34] both compile plan trajectory preferences from PDDL3.0 into Büchi automata and "simple preferences" into PDDL2.1 numerical fluents that are changed upon a preference violation. MIPS-XXL then uses Metric-FF with its enforced hill-climbing algorithm to find the final solution. On the other hand, MIPS-BDD stores the expanded search nodes in BDD form and uses a bounded-length cost-optimal BFS search for BDDs to solve the compiled problems. While compiling to NFA seems to allow those planners to handle the preference language PDDL3, it is not clear if there is any performance gain from doing so. Gamer, on the other hand uses a perimeter search, performing a breadth-first backward search to generate a pattern database for a later breadth-first forward search. To handle soft goals, the planner searches (without heuristic guidance) in a manner similar to our search, pruning nodes that appear worse in quality than the best-known plan. In HPlan-P, Baier et al. [5] compile trajectory preferences into additional predicates and actions by first representing them as a non-deterministic finite state automata (NFA). The heuristic

is then adjusted to take into account that different preferences have different values so that the planner is guided toward finding overall good quality plans. The planner is then extended to have a more sophisticated search algorithm where conducting a planning search and monitoring the parametrized NFA are done closely together [4].

Bonet & Geffner [12] present a planner whose search is guided by several heuristics approximating the optimal relaxed plan using the rank of d-DNNF theory. While the search framework is very similar to ours and the heuristic is relaxed plan-based, the problem tackled is a variation of PSP where goal utilities are not associated with facts achieved at the end of the plan execution but achieved *sometime* during the plan execution. This way, it is a step in moving from the PSP definition of traditional "at end" goals to a more expressive set of goal constraints on the plan trajectory defined in PDDL3. The heuristic computation they use is expensive, due to compiling the problem into a d-DNNF.

Compilation Approaches: While goal and action selection can be done directly during the search for plans, it is also possible to effectively compile out goal selection from the problem, as we saw in *Yochan*$^{\mathcal{COST}}$. This approach is quite appealing because any planner capable of handling action costs (along with whatever other constraints the problem may have) can be used to solve the problem. This effectively changes the search space representation, and while we saw this has a mostly negative effect in the comparison between *Yochan*$^{\mathcal{COST}}$ and *Yochan*$^{\mathcal{PS}}$, it allows the use of other planners so that no special mechanisms need be invented for goal selection. Indeed, Keyder & Geffner [66, 67] took this approach and show that it allows one to use the benefits of state-of-the-art planners. Their compilation differs from *Yochan*$^{\mathcal{COST}}$ in that (1) they do not handle conditional costs on actions

and (2) they use a compilation trick that forces the state-space into an "evaluation mode" such that costs for not achieving goals are only incurred during this mode. Using this compilation, they showed that planners made to solve partial satisfaction planning problems directly performed worse than current state-of-the-art cost-based planners. The advantage of using the latest techniques for cost-based planning is seductive, but it is unclear how well specialized techniques for goal selection would work in the state-of-the-art planners or how well they could handle more complex constraints on goal selection (e.g., goal utility dependencies or explicit resource limitations). Given our experiences with *Yochan*$^{\mathcal{COST}}$, it appears that handling soft goals directly can (at least in our framework) provide better quality plans.

These approaches are also unlikely to handle goal utility dependencies well when the heuristic is unable to take negative goal utility dependencies into account. Some preliminary experiments we have done have illustrated that this problem can occur when a heuristic simply ignores delete lists. This is because propagation and relaxed plan heuristics can assume that the positive-valued goal set can always be achieved together without penalty, and hence the heuristic will ignore negative goal utility dependencies associated with certain goal subsets.

Other compilation methods use solvers not explicitly made for planning problems. For instance, *OptiPlan* [95] extends an integer programming (IP) encoding for bounded parallel length classical planning to solve the PSP problem by adding action cost and goal utility. It relaxes the hard goal constraints by moving those goals satisfying conditions into the IP's objective function. This way, goals can be treated as soft constraints. The advantage of *OptiPlan*'s approach is that off-the-shelf IP solvers can be used to find the final plan that is guaranteed to be optimal up to a bounded parallel plan length. The disadvantage of this approach is that it

does not scale up well as compared with heuristic approaches, and one can see this in experiments on the encoding used for *iPUD*. van den Briel et al. [94] also proposed a set of constraints that could be applied to PDDL3. However, it appears this encoding was never implemented.

Another recent compilation approach tried by Russell and Holden uses a SAT encoding on PSP *net benefit* with goal utility dependencies [84]. It extends a version of the "thin-gp" encoding from SATPLAN [65], then encodes utilities using an objective function over a Weighted Partial Max-SAT (WPMax-Sat) problem. Like the *iPUD* approach, it is a bounded-length optimal encoding. In the problems generated by Russell and Holden, the approach scales nearly as well (and often better) than *iPUD*, though has markedly worse behavior in *zenotravel* as it extends its solving horizon. A somewhat similar SAT encoding was used for PDDL3 [53]. In both of these encodings, they first find the maximally achievable plan quality value C, then $n = \lceil log_2(C) + 1 \rceil$ ordered bits $b_1, ..., b_n$ are used to represent all possible plan quality values within the range of 0 to C. For the PDDL3-based planner, the SAT solver was modified with branching rules over those b_i bits. These are then used to find a bounded-length plan with the maximum achievable plan quality value.

7.3 SOLVING FOR QUALITATIVE PREFERENCES

Qualitative representations of preferences are typically non-numeric rankings between choices of goals (e.g., one might prefer white wine to red wine when one has fish). One problem with qualitative representations is that it is possible to generate different plans that are *incomparable* to one another (i.e., you cannot say whether they are better, worse, or of equal value). Nonetheless, they offer some advantages to users in that it is often easier for people to think symbolically rather than quanti-

tatively (e.g., saying one prefers white wine to red wine with fish can be easier than enumerating the possible values for each combination of wine and fish).

For the representation used by Brafman & Chernyavsky over TCP-Networks [16], a CSP-based planner is used to find a bounded-length optimal plan. They do this by changing the branching rules in a CSP solver so that the most preferred goal and the most preferred assignment for each goal are always selected first. Thus, the planner first branches on the goal set ordering according to goal preferences before branching on actions making up the plan.

Both logic-based [92] and heuristic search based [10] planners have been used to solve planning problems with qualitative preferences represented in the language PP by using weighting functions to convert qualitative preferences to quantitative utility values. This is due to the fact that quantitative preferences such as PSP and PDDL3 fit better with a heuristic search approach that relies on a clear way to compute and compare g (current cost) and h ("cost-to-go") values. The weights are then used to compute the g and h values guiding the search for an optimal or good quality solution.

7.4 Time-dependent Goal Costs

While temporal planning has long held the interest of the planning community (c.f., Zeno [80], TGP [90], TLPlan [1], Sapa [32], LPG [52], CRIKEY [26], TFD [38]), strong interest in preference-based and partial satisfaction planning (e.g., *net benefit* planning) is relatively recent.

My work on time-dependent goal costs can be seen as a cross-over between the areas. But others have emerged over the years. To our knowledge, the earliest work in this direction is by Haddawy & Hanks, in their planner Pyrrhus [55]. This planner allows a decision-theoretic notion of *deadline goals*, such that late

goal achievement grants diminishing returns [55]. For several years after this work, the topic of handling costs and preferences in temporal planning received little attention. As mentioned earlier, in 2006, PDDL3 [50] introduced a subset of linear temporal logic (LTL) constraints and preferences into a temporal planning framework. PDDL3 provides a quantitative preference language that allowed the definition of temporal preferences within the already temporally expressive language of PDDL2.1 [42]. However, few temporal planners have been built to support the temporal preferences available (c.f., MIPS-XXL [36], SGPLAN5 [64]), and none that are suitable for temporally expressive domains [27]. Other recent work uses the notion of time-dependent costs/rewards in continual planning frameworks (c.f., [73, 18]).

7.5 OTHER PSP WORK

We briefly go over some other related work on partial satisfaction planning, discussing partial satisfaction of numeric values, PSP *net benefit* using Markov Decision Processes (MDPs), techniques for oversubscribed scheduling and finally work related to our learning approach.

Degree of Satisfaction on Metric Goals: The reward models we have used have all dealt with logical goals. However, it is possible to specify reward on numeric values as well. Some of our previous work, done before beginning this dissertation work, handled numeric goal reward, where the definition of reward is over the final value of a numeric variable [8]. To handle this type of reward, we used a heuristic method similar to that of the planner Metric-FF, which effectively tracks upper and lower bounds on numeric variables on a planning graph structure. Using these bounds, it is then possible to estimate the cost (given through cost propagation) and reward for achieving certain values.

Using Markov Decision Processes (MDPs): Another way of solving PSP problems is to model them directly as deterministic MDPs [62], where actions have different costs. One way to look at this is to encode any state S (in which any of the goals hold) as a terminal state with the reward defined as the sum of the utilities of the goals that hold in S. However, rather than reifying goals rewards in this way, we can use a compilation approach similar to the one defined by Keyder & Geffner [67] discussed earlier, which avoids several problems (e.g., goal re-achievement) in the state space for the solving method. The optimal solution to the PSP problem can then be extracted from the optimal policy of this MDP. Given this, our solution methods can be seen as an efficient way of directly computing the plan without computing the entire policy (in fact, $h^*(S)$ can be viewed as the optimal value of S). For time-dependent costs or reward, it is also possible to formulate the problem using an MDP model [76].

Oversubscribed Scheduling: Over-subscription and partial satisfaction issues have received more attention in the scheduling community. Earlier work in over-subscription scheduling used "greedy" approaches, in which tasks of higher priorities are scheduled first [71, 82]. More recent efforts have used stochastic greedy search algorithms on constraint-based intervals [44], genetic algorithms [54], and iterative repairing techniques [72] to solve this problem more effectively. Some of those techniques can potentially help PSP planners to find good solutions. For example, scheduling tasks with higher priorities shares some similarity with the way $AltAlt^{ps}$ builds the initial goal set, and iterative repairing techniques may help local search planners such as LPG [51] in solving PSP problems.

Learning to Improve Plan Quality: There has been very little prior work focused on learning to improve plan quality. The closest learning system for planning

that tried to improve the quality of plans produced was the work by Pérez [81] almost a decade ago. In contrast to the approach in this dissertation, that work used explanation-based learning techniques to learn search control rules. As we discussed, one reason Stage-PSP outperforms SPUDS is that the S-SEARCH with learned evaluation function allows it to go to deeper parts of the search tree (and probe those regions with SPUDS search). While the Stage-PSP algorithm did not use the lookahead technique to reach deeper into the search space, this ends up achieving a similar effect.

7.6 PLANNERS USING IP OR LP IN HEURISTICS

This dissertation work makes extensive use of heuristics with embedded integer programming (IP) formulations. This allows the techniques to consider the complex interactions between goal and action selection in planning. Bylander [20] also used an IP formulation (and an LP relaxation) as a heuristic in the planner Lplan, but this heuristic has a bounded horizon, and so with action costs cannot be guaranteed optimal (unlike h^{GAI}_{LP} and h^{GAI}_{max}). Coles et al. [25] also have used LP formulations in combination with delete relaxation heuristics. However, their work focuses on increasing the informedness of heuristics for planning when there is an interaction between numeric variables. The work for embedded PDDL3 preferences into the planner OPTIC also uses IP formulations [21, 6]. Other work has used linear programming directly in the planning process to handle numeric [98] and temporal [75] aspects of the planning problem.

7.7 OTHER HEURISTICS USING FLOW MODELS

The structure encoding in our h^{GAI}_{LP} heuristic has strong connections to the *causal graph* [59] and *context-enhanced additive* [61] heuristics, both of which implement similar flow structure and procedurally solve the resulting relaxed model of the

planning problem. Indeed, both of these heuristics can similarly represent the negative interactions of actions and have shown better behavior when compared against a purely relaxed plan based heuristic in many domains. One difference with those heuristics, however, is that they are inadmissible whereas h_{LP}^{GAI} is admissible.

Chapter 8

Conclusion and Future Work

As agents acting in the real world, we must always make decisions on which sets of goals we should direct our actions toward achieving. Earlier work in automated planning addressed these issues, ranging from mention of the problem by Simon in 1964 [86] to more recent discussions on the subject with respect to decision theory [40, 55]. However, until recently work in the area has been sparse. This likely had to do with a lack of scalable methods for planning—it was hard enough to find short plans, let alone decide on which goals to achieve or the quality of the plans eventually found. However, now as we reach an era where automated methods for planning have become progressively more scalable and able to plug into larger, more complex systems, a user should naturally expect the ability to handle these real-life decisions on goal and constraint choice. Hence, it is imperative that the study of these types of problems progresses. This brings us to the main thrust of this dissertation work; that is, to expand upon representational and solving methods for partial satisfaction planning (PSP) problems. In particular, this work looks toward allowing a richer set of reward representations for handling goal choice. We defined *goal utility dependencies* and *time dependent goal costs* to these ends. For goal utility dependencies, we used the *general additive independence* (GAI) model. This model has the benefit that it fits well within heuristic search approaches, and it can be generated from other models. For time dependent goal costs, we presented a linearly increasing cost function after a deadline point, where penalty up to some maximum value would be given for failing to achieve a goal by a specified deadline.

We introduced several approaches used to solve for goal utility dependencies. First, we showed a novel heuristic framework that combines cost propagation and an integer program (IP) encoding to capture mutual dependencies of goal achievement cost and goal utility. We compared these heuristics to a bounded length IP-based solving method and found that, while the IP-based method often did well on easier problems, the heuristic method scaled much better. Of these methods, we found that the heuristic h^{GAI}_{relax}, which extracts a relaxed plan for all goals then encodes it in an IP along with goal utility dependencies, performed best among these methods.

After this, we introduced another novel heuristic based on a relaxed IP encoding of the original problem that keeps delete lists (unlike our other heuristics) but ignores action ordering. We then use the LP-relaxation of this encoding as an admissible heuristic and found that it performed better than h^{GAI}_{relax} and h^{GAI}_{max}, performing much better in terms of allowing us to reach optimal solutions, and finding better-quality solutions even when it did not lead to optimal solutions. Finally, we looked at a learning method based on the local search technique called STAGE with the intention of improving search.

We also explored temporal problems with *time-dependent goal costs*, or continuous cost functions that model certain classes of real-world problems with penalty costs for missing deadlines. We went on to show the advantages of reasoning with a continuous model of such problems versus a compilation to PDDL3 via discretization.

For future work on goal utility dependencies, it might be beneficial to use some of the recent work in partially including mutual exclusions in heuristics [58, 68]. Performed properly, this could allow us to only look at the mutual exclusions that

are specific to negative goal utility dependencies so that penalty might be avoided. Further, we intend to explore ways of integrating PDDL3 and continuous cost models, and supporting other continuous-cost measures, such as a continuous-cost analog to `always-within` (i.e., cost functions over time windows).

In terms of partial satisfaction planning generally, we plan to extend representational models to handle resource constraint issues. At a base level, one can view work in partial satisfaction planning as extending models of decision theory into the realm of planning. Indeed, early work in the area looked at the problem in this way and the use of general additive independence to model goal utility dependencies stems from decision theoretic work on preferences [2]. Work in handling partial satisfaction planning could be further enhanced by addressing issues of resource constraints, where resources that are not directly correlated with costs can be handled in conjunction with rewards for goals, as recently suggested by Smith [89].

[1] F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 16:123–191, 2000.

[2] Fahiem Bacchus and Adam Grove. Graphical models for preference and utility. In *In Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, pages 3–10, 1995.

[3] Christer Bäckström and Bernhard Nebel. Complexity results for SAS^+ planning. *Computational Intelligence*, 11(4):625–655, 1995.

[4] Jorge Baier, Fahiem Bacchus, and Sheila McIllraith. A heuristic search approach to planning with temporally extended preferences. In *Proceedings of IJCAI-07*, 2007.

[5] Jorge Baier, Jeremy Hussell, Fahiem Bacchus, and Sheila McIllraith. Planning with temporally extended preferences by heuristic search. In *Proceedings of the ICAPS Booklet on the Fifth International Planning Competition*, 2006.

[6] J. Benton, Amanda Coles, and Andrew Coles. Temporal planning with preferences and time-dependent continuous costs. In *Proceedings of the 22nd International Conference on Automated Planning and Scheduling*, 2012.

[7] J. Benton, Minh Do, and Subbarao Kambhampati. Anytime heuristic search for partial satisfaction planning. *Artificial Intelligence*, 173:562–592, April 2009.

[8] J. Benton, Minh B. Do, and Subbarao Kambhampati. Over-subscription planning with numeric goals. In *Proceedings of the Joint International Conference on Artificial Intelligence*, pages 1207–1213, 2005.

[9] J. Benton, Menkes van den Briel, and Subbarao Kambhampati. A hybrid linear programming and relaxed plan heuristic for partial satisfaction planning problems. In Mark Boddy, Maria Fox, and Sylvie Thiébaux, editors, *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS 2007)*, pages 34–41. AAAI Press, 2007.

[10] Meyghyn Bienvenu, Christian Fritz, and Sheila McIlraith. Planning with qualitative temporal preferences. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, pages 134–144, 2006.

[11] Avrim Blum and Merrick Furst. Planning through planning graph analysis. *Artificial Intelligence Journal*, 90:281–330, 1997.

[12] Blai Bonet and Héctor Geffner. Heuristics for planning with penalties and rewards using compiled knowledge. In *Proceedings of KR-06*, 2006.

[13] Blai Bonet, Gábor Loerincs, and Héctor Geffner. A robust and fast action selection mechanism for planning. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI 1997)*, pages 714–719. AAAI Press, 1997.

[14] Craig Boutilier, Fahiem Bacchus, and Ronen I. Brafman. UCP-networks: A directed graphical representation of conditional utilities. In *UAI*, pages 56–64, 2001.

[15] Justin Boyan and Andrew Moore. Learning evaluation functions to improve optimization by local search. *Journal of Machine Learning Research*, 1:77–112, 2000.

[16] Ronen I. Brafman and Yuri Chernyavsky. Planning with goal preferences and constraints. In *Proceeding of ICAPS-05*, 2005.

[17] Olivier Buffet and Douglas Aberdeen. FF+FPG: Guiding a policy-gradient planner. In *Proceedings of International Conference on Automated Planning and Scheduling*, pages 42–48, 2007.

[18] Ethan Burns, J. Benton, Wheeler Ruml, Minh B. Do, and Sungwook Yoon. Anticipatory on-line planning. In *Proceedings of the 22nd International Conference on Automated Planning and Scheduling (ICAPS)*, 2012.

[19] Tom Bylander. The computational complexity of propositional strips planning. *Artificial Intelligence Journal*, 69:165–204, 1994.

[20] Tom Bylander. A linear programming heuristic for optimal planning. In *AAAI-97/IAAI-97 Proceedings*, pages 694–699, 1997.

[21] Amanda Coles and Andrew Coles. LPRPG-P: Relaxed Plan Heuristics for Planning with Preferences. In *Proceedings of the 21st International Conference on Automated Planning and Scheduling (ICAPS)*, June 2011.

[22] Amanda Coles, Andrew Coles, Allan Clark, and Stephen Gilmore. Cost-sensitive concurrent planning under duration uncertainty for service level agreements. In *Proceedings of the 21st International Conference on Automated Planning and Scheduling (ICAPS)*, pages 34–41, June 2011.

[23] Amanda Coles, Andrew Coles, Maria Fox, and Derek Long. Forward-chaining partial-order planning. In *Proceedings of the 20th International Conference on Automated Planning and Scheduling (ICAPS)*, May 2010.

[24] Andrew Coles, Maria Fox, Keith Halsey, Derek Long, and Amanda Smith. Managing concurrency in temporal planning using planner-scheduler interaction. *Artificial Intelligence*, 173:1–44, 2009.

[25] Andrew Coles, Maria Fox, Derek Long, and Amanada Smith. A hybrid relaxed planning graph-LP heuristic for numeric planning domains. In *Proceedings of the 18th International Conference on Automated Planning and Scheduling*, pages 52–59, 2008.

[26] Andrew Coles, Maria Fox, Derek Long, and Amanda Smith. Planning with problems requiring temporal coordination. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI)*, July 2008.

[27] William Cushing, Subbarao Kambhampati, Mausam, and Dan Weld. When is temporal planning *really* temporal planning? In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1852–1859, 2007.

[28] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[29] Minh B. Do, J. Benton, Menkes van den Briel, and Subbarao Kambhampati. Planning with goal utility dependencies. In Manuela M. Veloso, editor, *Pro-*

ceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007), pages 1872–1878, 2007.

[30] Minh B. Do and Subbarao Kambhampati. Improving the temporal flexibility of position constrained metric temporal plans. In *International Conference on Automated Planning and Scheduling*, pages 42–51, 2003.

[31] Minh B. Do and Subbarao Kambhampati. Sapa: A multi-objective metric temporal planner. *Journal of Artificial Intelligence Research*, 20:155–194, 2003.

[32] Minh B. Do and Subbarao Kambhampati. Sapa: Multi-objective Heuristic Metric Temporal Planner. *Journal of Artificial Intelligence Research*, 20:155–194, 2003.

[33] Minh B. Do and Subbarao Kambhampati. Partial satisfaction (oversubscription) planning as heuristic search. In *Proceedings of KBCS-04*, 2004.

[34] Stefan Edelkamp. Optimal symbolic PDDL3 planning with MIPS-BDD. In *Proceedings of the ICAPS Booklet on the Fifth International Planning Competition*, 2006.

[35] Stefan Edelkamp and Malte Helmert. Exhibiting knowledge in planning problems to minimize state encoding length. In Susanne Biundo and Maria Fox, editors, *Recent Advances in AI Planning. 5th European Conference on Planning (ECP 1999)*, volume 1809 of *Lecture Notes in Artificial Intelligence*, pages 135–147, Heidelberg, 1999. Springer-Verlag.

[36] Stefan Edelkamp, Shahid Jabbar, and Mohammed Nazih. Large-scale optimal PDDL3 planning with MIPS-XXL. In *Proceedings of the ICAPS Booklet on the Fifth International Planning Competition*, 2006.

[37] Stefan Edelkamp and Peter Kissmann. Optimal symbolic planning with action costs and preferences. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, pages 1690–1695, 2009.

[38] Patrick Eyerich, Robert Mattmüller, and Gabriele Röger. Using the context-enhanced additive heuristic for temporal and numeric planning. In *Proceed-*

ings of 19th International Conference on Automated Planning and Scheduling (ICAPS), September 2009.

[39] Tom Fawcett. Knowledge-based feature discovery for evaluation functions. *Computational Intelligence*, 12:42–64, 1996.

[40] Jerome Feldman and Robert Sproull. Decision theory and artificial intelligence ii: The hungry monkey. *Cognitive Science*, 1(2):158–192, April 1977.

[41] Eugene Fink and Qiang Yang. Formalizing plan justifications. In *Proceedings of the Ninth Conference of the Canadian Society for Computational Studies of Intelligence*, pages 9–14, 1992.

[42] Maria Fox and Derek Long. PDDL2.1: An extension of PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003.

[43] Maria Fox and Derek Long. Modelling mixed discrete-continuous domains for planning. *Journal of Artificial Intelligence Research*, 27:235–297, 2006.

[44] Jeremy Frank, Ari Jonsson, Robert Morris, and David Smith. Planning and scheduling for fleets of earth observing satellites. In *Proceedings of Sixth Int. Symp. on Artificial Intelligence, Robotics, Automation & Space*, 2001.

[45] Angel García-Olaya, Tomás de la Rosa, and Daniel Borrajo. A distance measure between goals for oversubscription planning. In *Preprints of the ICAPS'08 Workshop on Oversubscribed Planning and Scheduling*, 2008.

[46] B. Gazen and C. Knoblock. Combining the expressiveness of ucpop with the efficiency of graphplan. In *Fourth European Conference on Planning*, 1997.

[47] Alfonso Gerevini, Yannis Dimopoulos, Patrik Haslum, and Alessandro Saetti. 5^{th} international planning competition website. `http://zeus.ing.unibs.it/ipc-5/`.

[48] Alfonso Gerevini, Patrik Haslum, Derek Long, Alessandro Saetti, and Yannis Dimopoulos. Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artificial Intelligence Journal*, 173(5-6):619–668, 2009.

[49] Alfonso Gerevini and Derek Long. Plan constraints and preferences in PDDL3: The language of the fifth international planning competition. Technical report, University of Brescia, Italy, August 2005.

[50] Alfonso Gerevini and Derek Long, editors. *Fifth International Planning Competition (IPC-5): planner abstracts*, 2006.

[51] Alfonso Gerevini, Alessandro Saetti, and Ivan Serina. Planning through stochastic local search and temporal action graphs in lpg. *Journal of Artificial Intelligence Research*, 20:239–290, 2003.

[52] Alfonso Gerevini, Alessandro Saetti, and Ivan Serina. An Approach to Temporal Planning and Scheduling in Domains with Predictable Exogenous Events. *Journal of Artificial Intelligence Research*, 25:187–231, 2006.

[53] Enrico Giunchiglia and M. Maratea. Planning as satisfiability with preferences. In *Proceedings of AAAI Conference on Artificial Intelligence*, pages 987–992, 2007.

[54] Al Globus, James Crawford, Jason Lohn, and Anna Pryor. Scheduling earth observing satellites with evolutionary algorithms. In *Proceedings of International Conference on Space Mission Challenges for Information Technology*, 2003.

[55] Peter Haddawy and Steve Hanks. Utility models for goal-directed decision-theoretic planners. *Computational Intelligence*, 14:392–429, 1993.

[56] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions of Systems Science and Cybernetics*, SSC-4(2):100–107, July 1968.

[57] Patrik Haslum. Additive and reversed relaxed reachability heuristics revisited. In *Booklet of the 2008 International Planning Competition*, 2008.

[58] Patrik Haslum. $h^m(P) = h^1(P^m)$: Alternative characterisations of the generalisation from $h^{\max}$ to h^m. In Alfonso Gerevini, Adele Howe, Amedeo Cesta, and Ioannis Refanidis, editors, *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, pages 354–357. AAAI Press, 2009.

[59] Malte Helmert. The Fast Downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[60] Malte Helmert. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence*, 173:503–535, 2009.

[61] Malte Helmert and Héctor Geffner. Unifying the causal graph and additive heuristics. In Jussi Rintanen, Bernhard Nebel, J. Christopher Beck, and Eric Hansen, editors, *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS 2008)*, pages 140–147. AAAI Press, 2008.

[62] Jessey Hoey, Robert St-Aubin, Alan Hu, and Craig Boutilier. SPUDD: Stochastic planning using decision diagrams. In *Proceedings of Proceedings of the Conference on Uncertainty in Artificial Intelligence*, pages 279–288, 1999.

[63] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[64] Chih-Wei Hsu, Benjamim Wah, Ruoyun Huang, and Yixin Chen. New features in SGPlan for handling preferences and constraints in PDDL3.0. In *Proceedings of the ICAPS Booklet on the Fifth International Planning Competition*, 2006.

[65] Henry Kautz, Bart Selman, and Jörg Hoffmann. Satplan: Planning as satisfiability. In *Booklet of the 5th International Planning Competition*, 2006.

[66] Emil Keyder and Hector Geffner. Set-additive and tsp heuristics for planning with action costs and soft goals. In *Proceedings of the Workshop on Heuristics for Domain-Independent Planning, ICAPS-07*, 2007.

[67] Emil Keyder and Hector Geffner. Soft goals can be compiled away. *Journal of Artificial Intelligence Research*, 36:547–556, September 2009.

[68] Emil Keyder, Jörg Hoffmann, and Patrik Haslum. Semi-relaxed plan heuristics. In *Proceedings of the 22nd International Conference on Automated Planning and Scheduling*, 2012.

[69] Jana Koehler. Planning under resource constraints. In *Proceedings of the 13th European Conference on Artificial Intelligence*, pages 489–493, 1998.

[70] Richard Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[71] Laurence Kramer and Mark Giuliano. Reasoning about and scheduling linked HST observations with spike. In *Proceedings of International Workshop on Planning and Scheduling for Space*, 1997.

[72] Laurence Kramer and Stephen Smith. Maximizing flexibility: A retraction heuristic for oversubscribed scheduling problems. In *Proceedings of IJCAI-03*, 2003.

[73] Seth Lemons, J. Benton, Wheeler Ruml, Minh B. Do, and Sungwook Yoon. Continual on-line planning as decision-theoretic incremental search. In *AAAI Spring Symposium on Embedded Reasoning: Intelligence in Embedded Systems*, 2010.

[74] D. Long and M. Fox. The third international planning competition (IPC3). http://planning.cis.strath.ac.uk/competition/, 2002.

[75] Derek Long and Maria Fox. Exploiting a graphplan framework in temporal planning. In *Proceedings of ICAPS-2003*, 2003.

[76] Mausam and Daniel Weld. Planning with durative actions in stochastic domains. *Journal of Artificial Intellgience Research*, 31:33–82, 2008.

[77] David McAllester and Robert Givan. Taxonomic syntax for first order inference. *Journal of the ACM*, 40(2):246–283, 1993.

[78] A. Newell and H. A. Simon. *Human problem solving*. Prentice-Hall, 1972.

[79] XuanLong Nguyen, Subbarao Kambhampati, and Romeo Sanchez-Nigenda. Planning graph as the basis to derive heuristics for plan synthesis by state space and csp search. *Artificial Intelligence*, 135(1-2):73–124, 2002.

[80] S. Penberthy and D. Weld. Temporal Planning with Continuous Change. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI)*, 1994.

[81] M. Alicia Pérez. Representing and learning quality-improving search control knowledge. In *Proceedings of the International Conference on Machine Learning*, pages 382–390, 1996.

[82] W. Potter and J. Gasch. A photo album of earth: Scheduling landsat 7 mission daily activities. In *Proceedings of SpaceOp*, 1998.

[83] R-Project. *The R Project for Statistical Computing*. www.r-project.org.

[84] Richard Russell and Sean Holden. Handling goal utility dependencies in a satisfiability framework. In *Proceedings of the 20th International Conference on Automated Planning and Scheduling*, pages 145–152, 2010.

[85] Romeo Sanchez-Nigenda and Subbarao Kambhampati. Planning graph heuristics for selecting objectives in over-subscription planning problems. In *Proceedings of ICAPS-05*, 2005.

[86] Herbert Simon. On the concept of the organizational goal. *Administrative Science Quarterly*, 9(1):1–22, June 1964.

[87] Herbert Simon. Motivational and emotional controls of cognition. *Psychological Review*, 74(1):29–39, January 1967.

[88] David E. Smith. Choosing objectives in over-subscription planning. In Shlomo Zilberstein, Jana Koehler, and Sven Koenig, editors, *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004)*, pages 393–401. AAAI Press, 2004.

[89] David E. Smith. Planning as an iterative process. In *Proceedings of the 26th AAAI Conference on Artificial Intelligence*, 2012.

[90] David E. Smith and Daniel S. Weld. Temporal Planning with Mutual Exclusion Reasoning. In *Procedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI)*, 1999.

[91] Edelkamp Stefan. Taming numbers and durations in the model checking integrated planning system. *Journal of Artificial Intelligence Research*, 40:195–238, 2003.

[92] Son Tran and Enrico Pontelli. Planning with preferences using logic programming. *Theory and Practice of Logic Programming*, 6(5):559–608, 2006.

[93] Menkes van den Briel, J. Benton, Subbarao Kambhampati, and Thomas Vossen. An LP-based heuristic for optimal planning. In Christian Bessiere, editor, *Proceedings of the Thirteenth International Conference on Principles and Practice of Constraint Programming (CP 2007)*, volume 4741 of *Lecture Notes in Computer Science*, pages 651–665. Springer-Verlag, 2007.

[94] Menkes van den Briel, Subbarao Kambhampati, and Thomas Vossen. Planning with preferences and trajectory constraints by integer programming. In *Proceedings of Workshop on Preferences and Soft Constraints at ICAPS-06*, 2006.

[95] Menkes van den Briel, Romeo Sanchez Nigenda, Minh B. Do, and Subbarao Kambhampati. Effective approaches for partial satisfaction (over-subscription) planning. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI 2004)*, pages 562–569. AAAI Press, 2004.

[96] Menkes van den Briel, Thomas Vossen, and Subbarao Kambhampati. Reviving integer programming approaches for AI planning: A branch-and-cut framework. In *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS-05)*, pages 310–319, 2005.

[97] Vincent Vidal. YAHSP2: Keep it simple, stupid. In *Proceedings of the 7th International Planning Competition (IPC'11)*, Freiburg, Germany, 2011.

[98] Steve Wolfman and Daniel Weld. The LPSAT engine and its applicationto resource planning. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 310–317, 1999.

[99] Sungwook Yoon, J. Benton, and Subbarao Kambhampati. An online learning method for improving over-subscription planning. In *Proceedings of the*

Eighteenth International Conference on Automated Planning and Scheduling, pages 404–411, 2008.

[100] Terry Zimmerman and Subbarao Kambhampati. Learning-assisted automated planning: looking back, taking stock, going forward. *AI Mag.*, 24(2):73–96, 2003.

APPENDIX A

ADMISSIBILITY OF h_{LP}^{GAI}

To show the admissibility of h_{LP}^{GAI}, we can show that h_{IP}^{GAI} captures a subset of the constraints in the original SAS+ planning problem. Since h_{IP}^{GAI} offers a further relaxation, it is also an admissible heuristic.

Theorem 1. *For large enough M, h_{IP}^{GAI} will always return solutions of value greater than or equal to the original planning problem P (that is, h_{IP} is an admissible heuristic).*

Proof. For the multi-valued representation of the planning problem, we can show that all feasible solutions to P can be mapped to feasible solutions of h_{IP}^{GAI}. Hence, h_{IP}^{GAI} is a relaxation of the original problem and is admissible when optimally solved given an objective function that returns a value equal to or greater than the original goal utility dependency planning problem.

The constraints produced by 4.13 and 4.14 help ensure that goal reward is counted appropriately in h_{IP}, and do not directly relate to the *feasibility* constraints in the original problem.

Recall that a solution to P, π, is a sequence of applicable actions starting from the initial state s_o. The mapping π to a solution the encoding for finding h_{IP} (a set of variable assignments in the integer program) is straight forward. First, assume all variables in h_{IP} are initially assigned to 0. For each appearance of an action in π, the corresponding variables in h_{IP} in are incremented by 1. That is, $\forall a_i \in \pi$ we increase the variable $action(a_i)$ by 1. We also increment action effect variables corresponding to $e_j \in effect_{a_i}$, such that $\forall e_j \in effect_{a_i}, effect(a_i, var(effect_{a_i,e_j}), e_j)$ is incremented by 1. Prevail conditions are handled similarly, where $\forall p_j \in prevail_{a_i}, prevail(a_i, var(prevail_{a_i,p_j}), p_j)$ is increased by 1. Goal utility dependencies and final state rewards are handled by taking the final state given from applying all actions in π (in order), s_n. For every

variable, $v \in V$ we take the value assigned to it in s_n, $f_{s_n} \in D_v$ and assign a value of 1 to $endvalue(v, f_{s_n})$. To handle goal utility dependencies, we take each dependency, G_k, and determine whether it is in s_n (a polynomial time operation). If so, then we assign a value of 1 to $goaldep(k)$.

This variable assignment scheme will always produce a feasible solution in h_{IP}. We show how each set of constraints is satisfied independently.

Equation 4.10: By definition of our translation, it is easy to see that the constraints generated by this equation will always be satisfied. The $effect(a_i, v, e)$ and $prevail(a_i, v, f)$ IP variables will be incremented if and only if $action_(a_i)$ is incremented. Hence, the constraints generated by equation 4.10 will always be satisfied.

Equation 4.9: Revisiting the definition of a feasible solution for P helps show how these constraints will always hold in our translation. Recall that a solution is feasible in P only if an action a_i can only be applied to a state s_i (i.e., a_i is applicable in s_i). One of the requirements for an action to be applicable is that its preconditions must hold in s_i. For that to be the case, one of two possible cases must be true. First, s_0 may have contained the assignment $v = f_j$ and no action prior to a_i has any effect (other than $v = f_j$) that changes v. Second, some action prior to a_i in the action sequence π, a_{i-x}, could have contained the effect $v = f_j$, and no other actions between a_{i-x} and a_i may have contained effects on v (other than $v = f_j$).[1]

Given our translation scheme, this would mean that constraints generated by equation 4.9 for value f on variable v would have a 1 on the left side of the equation if the first condition was met. Given the second condition, the effect variable on v

[1]These cases are intuitively easy to see and can be easily derived by induction.

for the action a_{i-x} becomes 1 on the left hand side (since a_{i-x} transitioned into the value f on variable v). Also, an effect variable for a_i becomes 1 on the right hand side. This means that, provided there always exists an effect that transitions from $v = f_j$, the right and left hand sides will always be equal.

Finally, to handle the case where no such transition from $v = f_j$ exists, we use the $endvalue(v, f_j)$ variable on the right hand side. This variable becomes 1 when s_n contains the assignment $v = f_j$. Similarly to action applicability, this occurs in two cases. First, when s_0 contains the assignment $v = f_j$ and no action in π contains an effect on v (other than $v = f_j$). Second, when an action a_{n-1-x} contained the effect assignment $v = f_j$ and no other action after a_{n-1-x} contains any assignment on v (other than $v = f_j$). This effectively models "end flow". Hence, the equations will always be balanced in our translation.

Equation 4.11: The left hand side is equivalent to the left hand side of equation 4.9. In the translation, IP variables associated with the prevail conditions of actions will always be increased by 1. Therefore, the prevail implication constraints will always be satisfied (with a large enough M value).

Equations 4.13 and 4.14: With the translation scheme, $goaldep(k)$ can only be 0 or 1. If goal dependency exists in the end state, s_n, then it has the value of 1. The end state values, $endvalue(v, f)$, are also binary in nature. They similarly can only be 1 if a particular value f is in the end state. To violate equation 4.13, the sum of all end values of a given dependency must be less than 1 despite the dependency existing. However, this cannot be the case because for the translation ensures that individual goal assignments within a goal utility dependency exist before increasing $goaldep(k)$.

Similar reasoning holds for equation 4.14. If a goal utility dependency exists (i.e., $goaldep(k) = 1$), then the associated end values (i.e., $endvalue(v, f)$) must have existed.

Objective Function: Since we have shown that h_{IP}^{GAI} is a relaxation of the original problem, we need now only show that the objective function allows the h_{IP} to return a value of greater or equal value to P. This is quite straight forward to see. The IP formulation is effectively equivalent to the objective of P, the maximization of net benefit. Therefore, when solved optimally it will always return a value equal to or greater than the optimal solution to P given that the problem is a relaxation of the original. □